source: https://doi.org/10.24442/boristheses.864 | downloaded: 5.5.2023

# Developing Ecosystem-aware Tools

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

**Boris Spasojević**

von Serbien

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik
Universität Bern

Von der Philosophisch-naturwissenschaftlichen Fakultät
angenommen.

Bern, 20.12.2016

Der Dekan:
Prof. Dr. Gilberto Colangelo

This dissertation can be downloaded from `scg.unibe.ch`.

Copyright ©2016 by Boris Spasojević

This work is licensed under the terms of the *Creative Commons Attribution – Noncommercial-No Derivative Works 2.5 Switzerland* license. The license is available at
`http://creativecommons.org/licenses/by-sa/2.5/ch/`

Attribution–ShareAlike

*To the girl who waited.*

# Acknowledgements

I warmly thank anyone and everyone who directly or indirectly contributed to this work. To make this more personal, let's name some names:

First, I'd like to thank Oscar Nierstrasz for too many things to list, from guidance in my research to after-lunch puzzles. Oscar is a big part of what made my PhD student life as good as it was.

I am very thankful to Mircea Lungu, his input was crucial for my work, and his optimism would often reignite my interest in my own work. He is by far the most positive person I know, and I hope that never changes.

I am grateful to Coen de Roover for his feedback on my work, as well as for agreeing to be on the PhD committee. I am also grateful to Matthias Zwicker for charring the committee.

A big shout out to all the PhD students of the Software Composition Group that have ever given a weekly report on a Tuesday, especially the ones that did it while I was there: Erwan Wernli (a.k.a. the train man), Niko Schwarz (and his shadow), Andrea Caracciolo (Caracciooooolooooo), Andrei Chiş (a.k.a. the moldable gipfeli man), Jan Kurš (did you Czech that?), Haidar Osman (a.k.a. the bug-predicting fat man) Nevena Lazarević (a.k.a. the other Serb), Leonel Merino (is it Chile in here?), Yuriy (salami) Tymchuk, and Claudio Corrodi (a.k.a. the fake Italian). You guys made SCG awesome and I'm honored to have shared the SCG staff page with you! I'd also like to thank Mohammad Ghafari for his input and insistence on related work. A very special thanks goes to Iris Keller for all the help and for supporting me in learning German.

I'm thankful for the support of my parents and I hope I made them proud. Moving to Switzerland was made so much easier knowing that Peđa and Mima are nearby, but even if I was living on Mars my first phone call would be to them. I'm thankful to Ivana for her patience and for making me a better human being. Also, I'm very

happy I get to say thanks to baby Sana!

I'd like to thank all my friends from Novi Sad for making me feel at home whenever I was there. They are, in no particular order: Peđa Jovanović, Nikola Živković, Darko Pejaković, Slobodan Prljević, Vladan Marsenić, Andrea Pavlović, Radovan Prodanović, Nikola Prostrog, Darja Krek, Boban Magoč, Ivan Dobrić and Ana Marija Ćirić. Atila and Danka Pekter as well as Sofija Hotomski and Slobodan Adamović get the same gratitude for Münich and Zürich.

Finally, I'd like to thank the people that introduced me to the world of compilers and program analysis and guided me academically before I embarked on my PhD studies: Zorica Suvajdžin, Miodrag Đukić, Zoran Zarić, Momčilo Krunić and Marko Krnjetin.

# Abstract

Tool developers frequently leverage data from software ecosystems to improve their tools. Unfortunately, every developer has to build his own infrastructure to analyse the software ecosystem. This means identifying the scope of the ecosystem, obtaining the source code, extracting, storing and updating the data and so on.

We argue that many of these tasks can be automated, freeing the developer to focus only on how to extract the needed ecosystem data and how to present it to the developer.

To support our claim, we developed a framework for developing ecosystem-aware tools, tools that leverage data from the software ecosystem. This framework automates all routine steps of the process and leaves the developer to specify what data to extract from the ecosystem, and how to use it.

To illustrate how this framework can be used for development of real-world ecosystem-aware tools we created four such tools using this framework. These tools are implementations of innovative approaches that improve the developer experience and were chosen to be diverse so as to illustrate the flexibility and features of the framework which is meant to support the needs of a broad range ecosystem-aware tools.

The tools are individually evaluated and shown to be an improvement on the standard techniques, further supporting the notion that incorporating ecosystem data into the development process can be beneficial.

# Contents

# 1 Introduction

Software systems do not live in isolation. Whether through direct dependencies or shared libraries all software systems share some sort of connection with other systems. These systems co-evolve as changes in one project cause changes in its downstream dependencies, changes in libraries ripple through their clients *etc.* We call a group of software systems united by common or mutual dependencies a software ecosystem.

Many aspects of Software Engineering have made great progress in embracing the concept of software ecosystems in one way or another. For example, modern software build systems such as Maven make dependencies between projects explicit. This also defines relations between projects and inadvertently defines the scope of an ecosystem forming around particular libraries and framework. Tools like this enable us to get the scope of a software ecosystem very quickly.

On the other hand, creators of development tools have realised that projects in the same ecosystem tend to share certain traits, and that identifying these traits can help developers during software engineering tasks. The idea is that if developers of related projects did things a certain way, there is a strong probability that other

developers, working in the same ecosystem, should do these things in the same way. This simple idea spawned many directions of research into how data from related projects can be integrated in the software development process. We qualify as "Ecosystem-aware" any software development tool that in any way incorporates data from the software ecosystem.

This large body of work is unfortunately completely fragmented, with practically every tool relying on data extracted from a different set of projects, using custom infrastructure consisting roughly of a sequence of steps shown in Figure 1.1. These infrastructures essentially re-implement the same functionality for each individual tool with little or no concern for re-use by other developers. Another negative side effect of this is that it is more difficult for tool developers to communicate the way their tools work, as one must understand much about how, and from where, the data is gathered rather than what data is gathered and why.

Another shortcoming of these custom infrastructures is that they provide little or no support once the data that is gathered is stale. This reduces the life span of the tool, as stale data will most likely reduce the quality of the tool that relies on it.

We argue that much of the process of ecosystem-aware tool development can be automated and abstracted away from the tool developer. We partly draw inspiration from distributed data analysis platforms such as Hadoop or Spark which allow a developer to write how the data is to be processed and not where each part of the data can be found and how the analysis should be run. The framework is in charge of the routine parts of the process, and the developer is not burdened by non-important details regarding the execution of the analysis. Similarly, if the scope of the software ecosystem of interest to the developer is defined, many of the steps of gathering ecosystem data can be part of a framework rather than the job of the developer. These steps include obtaining the source code of the projects in the ecosystem, executing user specified pre-processing, analysis and post-processing, storing the data for later use *etc.* Figure 1.2 shows the same steps from Figure 1.1 but now divided into the ones that can be automated and ones that can not.

As a proof of this concept we implemented a framework for developing ecosystem-aware tools for Pharo Smalltalk[1]. We call it the "Ecosystem Monitoring Framework" or EMF for short.

[1] `http://www.pharo-project.org`

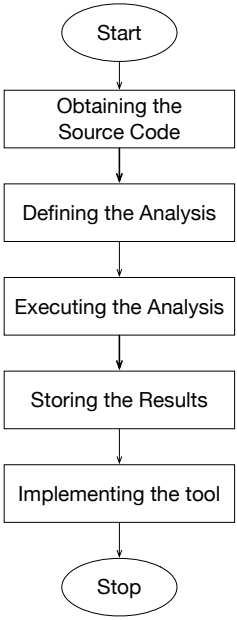


Figure 1.1: Steps needed to develop an ecosystem-aware tool.

This framework defines the scope of the Pharo Smalltalk ecosystem, and provides a set of APIs and configuration mechanisms that enable the developer to specify all steps for gathering the needed data. Furthermore, the framework automatically keeps this data up to date by periodically re-analyzing the ecosystem. It also provides a centralised place to store and read the data.

To illustrate how this framework can be used for development of real-world ecosystem-aware tools we created several such tools using this framework. These tools were chosen to be diverse so as to illustrate the flexibility and features of the framework which is meant to support the needs of a broad range of ecosystem-aware tools.

The first tool implemented is a type inference engine, leveraging ecosystem data on how often methods are invoked on instances of types to sort candidate types of a variable. This is the canonical ecosystem-aware tool developed with EMF. To illustrate that EMF

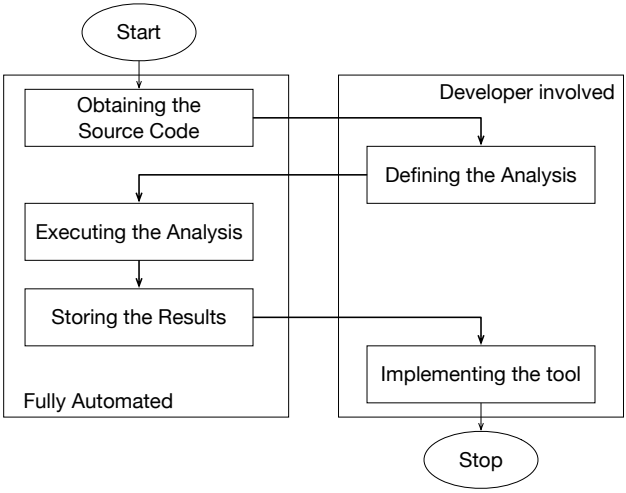


Figure 1.2: Steps needed to develop an ecosystem-aware tool, separated by whether or not they can be fully automated.

decouples gathering data from presenting it, we augmented a code browser to show frequently used methods of the current class by re-using the data gathered for the ecosystem-aware type inference for a different use case. Since EMF analyses the entire ecosystem, we can use it to study the state of the ecosystem in order to make data-driven decisions. We illustrate this by exploring the state of the method argument names in the ecosystem in order to improve a heuristics based type inference technique called type guessing. And finally, since EMF is not limited to static analysis we implemented an approach to provide tools with objects on demand, by mining code snippets whose execution produces objects.

## 1.1 Thesis Statement

We formally state our thesis as follows:

*Routine parts of ecosystem-aware tool development can be automated, providing developers with a framework for tool development focused only on analysis and presentation of ecosystem data.*

This thesis statement raises several scientific challenges with regards to identifying the scope of the ecosystem, providing a suitable interface to the developer for analyzing the projects in order to support ecosystem-aware tools, solving the problem of data staleness *etc.*

## 1.2 Contributions

This section outlines the main contributions of this thesis. They can roughly be divided in two parts: The idea and proof of concept implementation of the framework for development of ecosystem-aware tools and the innovative tools developed using this framework. The contributions consist of both the implementations as well as the scientific publications that resulted from the studies conducted to motivate or evaluate the tools.

### 1.2.1 Ecosystem Monitoring Framework

Integrating ecosystem data into developer tools can be very beneficial but is usually complicated. Our thesis is that automating the routine parts of this task can reduce the amount of work needed to develop these tools. To support this claim, we have developed a framework that allows developers to quickly develop new tools that use ecosystem data. We call this framework the "Ecosystem Monitoring Framework" or EMF for short. The framework automates the execution of user-defined analyses on ecosystem projects, allowing the developer to focus only on what ecosystem data is needed for her tool and how to present it. This is, to the best of our knowledge, the first attempt to implement a framework for simpler development of ecosystem-aware tools, and the fact that our implementation of the framework was robust enough to build the tools that we did is a testament to the versatility of the idea of a framework for ecosystem-aware tool development.

### 1.2.2 Ecosystem-aware Tools

This section gives a short description of the four ecosystem-aware tools developed using the ecosystem monitoring framework. Each of these tools is a proof of concept implementation of an idea on how one could improve the development process by including ecosystem

data. Each tool also illustrates one important aspect of the framework. These aspects, as well as their implications are discussed as part of the description of each tool.

### Improved Type Inference for Dynamically Typed Languages

Dynamically-typed languages lack information about the types of variables in the source code. Developers care about this information as it supports program comprehension. Basic type inference techniques are helpful, but may yield many false positives or negatives.

In ecosystem-aware type inference we track how often messages are sent to instances of available types throughout the source code for all available projects from the ecosystem. This means that the back end of the ecosystem-aware type inference builds a weighted mapping from classes to selectors, where the weight is the number of times a message with that selector was sent to an instance of that class.

We use this information to sort the potential types of a variable based on their likelihood of being the actual type in the context. The likelihood is computed based on how many times the messages sent to this variable have been observed to be sent to each potential type throughout the ecosystem.

Using EMF, we implemented a prototype and used it to evaluate the approach. We show that, for our implementation, measuring the frequency of association between a message and a type throughout the ecosystem source code is helpful in identifying correct types.

Results of this work were published in the proceedings of an international conference [SLN14a].

This tool illustrates the standard way of developing ecosystem-aware tools using EMF. It uses EMF to define and execute the ecosystem analysis and has a simple front end for developers to interact with.

### Documentation and Code Browser Augmentation

Software developers are often unsure of the exact name of the method they need to use to invoke the desired behavior in a given context. This results in a process of searching for the correct method name in documentation, which can be lengthy and distracting to the developer.

We can decrease the method search time by enhancing the documentation and code browser of a class with the most frequently used methods. Usage frequency for methods is gathered by analyzing other projects from the same ecosystem.

Using EMF, we implemented a proof of concept of the approach. We use the same data gathered for the ecosystem-aware type inference, but a different front end. Since methods are commonly searched for using a system browser called "Nautilus", we developed a plugin for it which adds a section with the most commonly used methods for the currently observed class.

Results of this work were published in the proceedings of an early research achievements track of an international conference [SLN14b] and an international workshop [SLN14c].

This tool illustrates the clear distinction between gathering the data and presenting it to the developer. This tool uses the exact same data as the ecosystem-aware type inference, gathered by the same analysis, and presents it to the developer in a different way to obtain a different result. This shows that the two concerns of the developer (data analysis and presenting the data) can be decoupled.

### Improved Type Guessing for Method Arguments

A common practice when writing Smalltalk source code is to name method arguments in a way that hints at their expected type (i.e., *aString*, *anInteger*, *aDictionary*). This practice makes code more readable and some tools (such as the auto complete feature in the Pharo Smalltalk code editor) improve the developer experience by "guessing" the type of the method argument based on these hints.

Using EMF we gather argument names throughout the ecosystem and generate a weekly report containing information about commonly used ones. This report can be used by the Type-guessing tool developer to include heuristics for better type guessing. Using these reports we developed heuristics that improved the Pharo type-guesser by almost 40%.

Results of this work were published in the proceedings of an international conference [SLN16].

This tool shows that it is possible to use EMF as a platform for writing analyses that provide a report on the state of the ecosystem and don't necessarily have a standalone front end. In this case, we used EMF to analyze the state of argument names throughout the

ecosystem and used the generated report to make informed decisions on how to improve type guessing heuristics.

### The Object Repository

The Object Repository is just a back end for multiple potential front ends. The main idea behind the Object Repository is to enable front end tools to have "Objects on demand". This means that the Object Repository mines, from the ecosystem, code snippets that, when executed, produce an instance of some class. These snippets are then stored and mapped to the class they can instantiate. A front end needs only to provide a class name, and the Object Repository will provide all available snippets that instantiate that class. These snippets have many potential use cases in software documentation, testing, program comprehension *etc.*.

Our proof of concept implementation of the Object Repository relies on brute force execution of code segments obtained by converting AST nodes of methods to source code. We show that applying the proposed approach to the Pharo ecosystem, as defined in EMF, results in an Object Repository that can instantiate almost 80% of the available classes in these projects.

Results of this work were published in the proceedings of an international workshop [SGN16].

This tool shows that EMF is not limited to just analyzing the source code as text, but also supports advanced techniques such as execution of code, which opens many other doors for ecosystems-aware tools.

## 1.3 Outline

The main chapters of this thesis present the ecosystems monitoring framework and the ecosystems-aware tools developed on top of it. The thesis is thus organised as follows:

**Chapter 2** provides a survey of the literature focusing on work that is related to software ecosystems, large-scale software analysis and using ecosystem data in the development of developer tools.

**Chapter 3** presents a technical description of the requirements and implementation of the ecosystems monitoring framework. This

chapter also provides an overview of how to develop a toy ecosystem-aware tool.

**Chapter 4** presents the ecosystem-aware type inference engine.

**Chapter 5** presents the improved system browser augmented with ecosystem data, as well as the study that motivated the augmentation.

**Chapter 6** presents the improvements to the type guessing system in Pharo that where made possible by a reporting tool built with EMF.

**Chapter 7** presents a framework that used EMF to mine the ecosystem for code snippets which, when executed, produce objects. These snippets can be used for several different software engineering tasks.

**Chapter 8** concludes the thesis and outlines open questions.

# 2
# State of the art

In this chapter we survey the state of the art in three fields directly related to this thesis. These fields are software ecosystems, platforms for large scale software analysis and ecosystem-aware tools.

## 2.1 Software Ecosystems

The concept "Software ecosystem" was introduced by Messerschmitt and Szyperski in 2003 [MS03] and has since then spawned a wide research field with, as reported by a longitudinal study by Manikas [Man16], two dedicated workshops (which have since merged), a major presence at conferences and 231 publications over 8 years.

There are multiple definitions of a software ecosystem throughout literature, and we provide a chronological review of the definitions for a software ecosystem, and discuss how the usage of the term in this thesis reflects on these existing definitions.

We start with the definition by Messerschmitt *et al.* :

*"Traditionally, a software ecosystem refers to a collection of software products that have some given degree of symbiotic*

*relationships."* [MS05].

This definition is quite vague, so our usage of software ecosystem falls completely within it, since we assign the symbiotic relationships to be shared or mutual dependencies and co-evolution.

Jansen *et al.* provide a very different definition:

*"We define a software ecosystem as a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently underpinned by a common technological platform or market and operate through the exchange of information, resources and artifacts."* [JFB09]

This definition is oriented more towards the business world *i.e.,* commercial entities and software ecosystems they interact with. Examples could include the Google Play store for Android apps or the Apple App Store for iOS apps. This is a substantially different view of software ecosystems, as it includes not just the software but also the legal entities that shape the ecosystem. The main similarity to our usage of the term is the observation that these ecosystems are frequently underpinned by a common technological platform, which is also true in our use of the term.

Bosch *et al.* define a software ecosystem as follows:

*"A software ecosystem consists of a software platform, a set of internal and external developers and a community of domain experts in service to a community of users that compose relevant solution elements to satisfy their needs."* [BBS10]

While we do focus on a particular software platform (Pharo Smalltalk) our focus is not at all on the community or users of the solutions built using this platform. While this definition opens more opportunities for gathering data from alternative sources (mailing lists, Q&A forums, bug reports, *etc.*) this thesis is limited, for simplicity, only to the source code of the solutions mentioned in the definition.

The definition provided by Lungu *et al.* most closely matches our own:

*"A software ecosystem is a collection of software projects that are developed and evolve together in the same environment."* [LLGR10]

This definition, similar to the one by Messerschmitt, is quite vague. If we consider the environment to be the platform and commonly available libraries used for development, then this definition fits ours nicely. We just aimed to be a slightly more specific in defining the term.

Finally, we look at the definition by Manikas *et al.* :

*"A software ecosystem consists of a group of actors, one or more business models that serve these actors in a possible wider sense than direct revenues, one or more software platforms that the business models are built upon and the relationships of the actors and business models."* [MH13]

This definition is a somewhat improved version of the definitions provided by Jansen and Bosch. It suffers from the same points distancing it from our definition. While the definition does include a software platform and the software (business models) it gives an explicit inclusion to the actors and their relations as part of the ecosystem. As mentioned before, including this, while very desirable, escapes the scope of this thesis.

Finally, we reflect on our definition of a software ecosystem. It states:

*"A software ecosystem consists of a group of software systems united by common or mutual dependencies."*

We prefer this definition as it makes explicit our objects of interest — the software systems — as well as the way they are interconnected — common and mutual dependencies.

## 2.2 Platforms for Large-scale Software Analysis

In this section we present an overview of existing platforms for large-scale software analysis. This field is motivated by the large research interest in mining software repositories, and deals with

analysis of source code, metadata and software evolution. Most of the approaches use domain specific or query languages to run the analyses, which can be a burden for developers and they neither provide automated updating of data nor a centralised repository for the results.

Since mining software repositories is very often concerned with studying software evolution, several platforms have been developed with special focus on software evolution. Tools such as Minero [ADG04] and Bloof [DP03] provide database inspired approaches for analyzing the history of a project relying on query languages. These approaches are quite limiting as the project data is pushed into a rigid schema keeping only the information specified by the schema thus excluding certain analyses. On the other hand, Kenyon [BWKG05] presents a much more flexible framework which enables the developer to specify the required information through ORM mappings. This data is later available for post processing by external evolution analysis tools.

Tools such as these analyse large quantities of historical data about only one project. Since throughout this thesis we do not focus on the history of our projects, these approaches are of limited applicability.

The Pangea [CCSL14] platform offers a way to analyse predefined software corpora in a language agnostic way. It also uses a domain specific language based on the Moose [NDG05] analysis platform, which is essentially an advanced query language for the FAMIX meta model. The main shortcoming of Pangea is that it deals with software corpora rather than ecosystems. By this we mean that the corpora are frozen in time *i.e.,* the projects are fixed to a certain version, and as the projects evolve the data gathered would be stale. Also, Pangea has no central way of storing the results, or providing them to clients.

Sourcerer [BOL09] is a research project aimed at exploring open source projects through the use of code analysis. The infrastructure is for automated crawling, parsing and storage of open source software at scale. Sourcerer relies on analysing the code in order to extract textual information as well as structural and metadata information mainly used to improve code search. They use standard text retrieval techniques combined with source code specific heuristics to produce a relational representation of the source code which they claim is suitable for applying machine learning techniques for

a variety of tasks. Sourcerer suffers from many of the same shortcomings as Pangea does. Primarily, it works with data sets rather than live projects, and uses a very specific relational representation of the gathered data.

Boa [DNRN13] is a domain specific language with an accompanying infrastrucure for ultra-large-scale software repository and source code mining. By leveraging distributed data processing techniques provided by Hadoop [Whi12] and the Map-Reduce [DG08] paradigm Boa is able to process petabytes of software repository data. To protect the developer from expressing the analysis in the Map-Reduce paradigm the authors implemented the Boa language that includes domain specific types such as `Project`, `CodeRepository`, `Revision` and `Person` and enables the developer to write procedural code that gets translated to Map-Reduce jobs. This code is either query like in nature, or consists of a series of AST visitors that extract the required data. Boa is accessed through a web interface where users can submit jobs which will eventually run on the infrastructure. Since Boa is build on Hadoop, the results of the analyses are stored in the Hadoop file system but are not available for front-end clients on demand. Also, the Boa infrastructure is not available so users cannot run Boa on their own data cluster. Furthermore, Boa includes a wide range of projects, including their entire histories, written in different languages so identifying the scope of an ecosystem would fall on the user.

Finally, it should be noted that none of these platforms support dynamic analysis in any way, thus ecosystem-aware tools such as the Object Repository could not be implemented using these platforms.

## 2.3 Ecosystem-aware Tools

A large body of work exists in the field of ecosystem-aware tools. Authors generally do not use the label "ecosystem-aware" for their tools, and more often refer to "mining projects" or "mining software repositories". This is because most authors ignore software evolution and extract the data for their tool from a static snapshot of a large body of related projects. Nevertheless, all of the approaches presented in such papers could be implemented using a variation of the ecosystem monitoring framework, which would address this issue. Thus, we label all tools that leverage data from a large body

of related projects as "ecosystem-aware".

These tools cover a variety of software development issues: from predicting which parts of the system are likely to have defects [DLR11], to automatically detecting code clones across open source systems [RCK09], supporting code search across the web or large collections of software projects [BOL14] and automatically detecting the license of jar archives [DPGA10].

We can roughly divide these tools in several categories, each with a corresponding subsection.

### 2.3.1 API-Related Tools

Research into tools and analyses of API usage directly relates to the frequently used methods tool presented in Chapter 5. This tool was implemented to demonstrate the disconnect between the front end and back end of ecosystem aware tools, but any of the techniques present in the literature could be used to build a tool using EMF.

One direction of research related to ours is the work on API specification mining [RBK+13]. In order to detect groups of methods that are usually called together Nguyen *et al.* [NNP+09] statically analyse method call and field access graphs. The mined API patterns represent sets of methods that are called on a single object. Pradel and his colleagues used a combination of static and dynamic analysis to automatically detect illegal uses of APIs [PJAG12] while building multi-object protocols.

Tools such as MAPO [ZXZ+09] focus on mining API methods that are frequently called together and their usages follow sequential rules. Other approaches such as those presented by Buse *et al.* [BW12] focus on a different kind of static analysis based on combination of path sensitive dataflow analysis, clustering, and pattern abstraction. A tool called PROSPECTOR [MXBK05] introduces the concept of *jungloid* source code in an attempt to simplify the mined snippets in order to enable synthesis and combining to form more complex code fragments.

In their work, Thummalapenta and Xie crawl the web for methods and classes that are often reused [TX08]. They collect the frequency of calls to methods and classes of a given API in order to recognize so-called hotspots. The information that they collect is similar to ours; the difference is in the end-goals — unlike their, and most of the related work which is targeted at Java systems, we

have to address challenges inherent to dynamically typed programming languages.

De Roover *et al.* aim to help understanding of APIs for both providers and users with a tool called EXAPUS [RLP13]. It is an IDE-like tool that enables exploration with respect to multiple dimensions (hierarchically organized scopes of projects and APIs, API usage metrics, API metadata, project-centric versus API-centric views). They exercise their tool with this tool is their QUAATLAS [RLP13] corpus. It is a corpus for API-usage analysis built on top of the existing QualitasCorpus [TAD+10] by revising it so that fact extraction can be applyed at a level suitable for API-usage analysis.

### 2.3.2 Code Examples

A common approach to help developers understand certain aspects of code is to provide an example of the explained phenomenon. This is a common occurrence in software documentation where the examples are normally provided manually by the documentation author. Due to the additional effort needed to provide these examples, much work has been done in an attempt to mine examples from existing source code.

Examples are most commonly mined from open source repositories [ZXZ+09, HM05, MXBK05] but other sources are also used *e.g.,* Google code search [TX07]. Several approaches are used to present the code examples to the developer. These include search engines [TX07], IDE augmentations [ZXZ+09], adding code examples to documentation [MBFV13] and others.

Multiple approaches have been proposed to mine relevant code examples and use them to improve code completion tools. Kim *et al.* propose generating so called eXoaDocs or example oriented API documents which are used to supplement API documentation with examples [KLHK09]. Holmes *et al.* propose Strathcona in an attempt to minimise the effort for the developer to query for examples [Hol06]. Bruch *et al.* [BMM09] explore three different strategies for using information gathered from existing code repositories. Their approach, given a set of methods that have been called on a variable and the enclosing method as context, recommend missing method calls for that variable. The PARSEWeb tool [TX07], rather than performing the analysis of open source systems itself, is build atop

of code search engines in order to try to generalise their approach. Pavlinovic *et al.* [PB13] mines code examples that occur more than a given threshold in a given code repository, and provides relevant examples on demand and taking into account the developer's current context. Zhang *et al.* focus on automatically filling the parameter list of API calls automatically [ZYZ+12].

Other work, such as that of Ghafari *et al.* [GGMT14] focuses on mining examples from unit tests claiming that this is a good source of examples as they are concise, relevant and trustworthy.

### 2.3.3 Inter Project Dependencies

Ossher *et al.* present a method for automatically resolving dependencies for open source software which works by cross-referencing the missing type information in a project with a repository of candidate artifacts [OBL10]. They build a detailed AST for every individual system they analyze. The AST is used to analyze an individual project and detect the missing types. Once the missing types have been identified, the final step is to match them against the artifacts in the candidate repository.

Similarly, Lungu *et al.* have also analyzed inter-system dependencies by analyzing the method call graph, identifying the targets of calls which do not exist inside of a system, and detecting these targets elsewhere in the ecosystem [LRL10].

Approaches such as these would be very useful if implemented as an ecosystem-aware tool since resolving and obtaining all the required dependencies to build a software system is not a trivial task.

Decan *et al.* take this idea further by exploring package dependencies in an inter repository context. Namely, the R ecosystem had one main repository in use (CRAM) but the rising popularity of GitHub has produced a divide in the ecosystems. Decan *et al.* explore the impact of GitHub on the R ecosystem as well as the challenges of inter-repository package management [DMCG16].

Mileva *et al.* mined software repositories to observe popularity of APIs, and help developers choose a popular API rather than an unpopular one, claiming that the "wisdom of the crowd" can be used to measure the quality of an API [MDZ10].

Kula *et al.* visualize the evolution of the relation between a system and its dependencies using two different views [KRG+14]. The

system-centric dependency plots visualizes the successive library versions that a system depends on over time enabling insight into changes in dependencies along the release history of the system. On the other hand, the library-centric dependants diffusion plot shows the diffusion of users across different versions of the library. These visualizations help both the developers of a library as well as its clients.

# 3

# Ecosystem Monitoring Framework

## 3.1 Introduction

Leveraging ecosystem data in software development by including it in developer tools can benefit software quality and speed of development in many ways. A large body of work already exists dealing with integrating ecosystem data *i.e.,* data from other related projects, into the development process. Examples of such work deal with code completion [BMM09], method and argument recommendations [NNP+09, AXPX07], library scoring systems [MDZ10], code snippet recommenders [ZXZ+09, HM05, MXBK05, TX07] and others.

Papers published in this field rarely take into account software evolution. They focus heavily on the usability and helpfulness of their respective tools and provide no support for the tool once the data it is based on is out of date. However, tool developers need tool support to easily gather ecosystem data and keep the data up to date.

We implemented one such tool for the Pharo Smalltalk ecosystem and named it the "Ecosystem Monitoring Framework" or EMF. The main goal of this framework is to enable developers to write

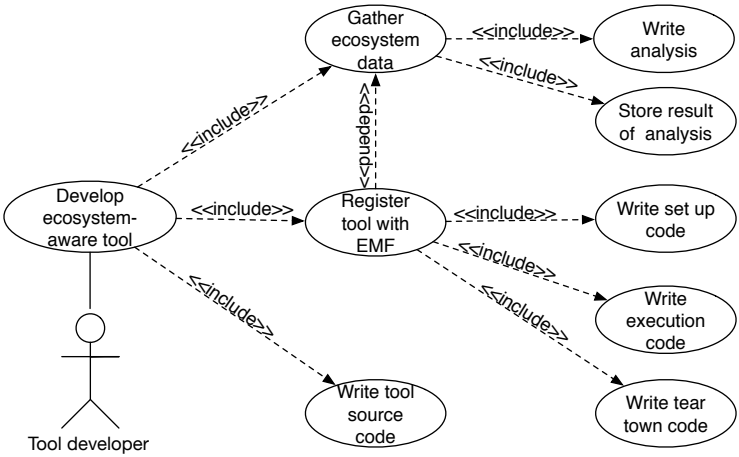


Figure 3.1: Use case diagram of a tool developer using the ecosystem monitoring framework.

tools that leverage ecosystem data (*i.e.,* ecosystem-aware tools) without requiring them to spend time on the infrastructure of running their analyses or keeping the resulting data up to date.

## 3.2 EMF Requirements

To guide our decision making process in developing EMF we define a list of requirements for the system. These requirements are informed by the desired use case scenario for EMF shown in Figure 3.1. This system has only one user, a tool developer, with one associated use case — developing a tool that relies on ecosystem data.

Developing an ecosystem-aware tool can be separated into three tasks:

1. Gathering the required ecosystem data.
2. Writing the source code for the tool that uses that data.
3. Registering the ecosystem aware tool with an instance of the framework.

Gathering ecosystem data is further broken down into writing the analysis to gather the data, and storing the results. Registering

the tool with EMF consists of defining how to set up, run and tear down the analysis. The framework, aware of these steps, should execute this analysis on all projects in the context of the ecosystem. In short, we want the developer to specify how to get the data she needs, and how to provide it to the developer. The developer should not have to focus on the technicalities of execution across projects.

With these use cases in mind, we first eliminate those that are independent of the framework *i.e.,* we eliminate all use cases that can be fulfilled by the user without interacting with the framework. Firstly, writing the source code for the tool is solely the developer's responsibility and in no way influences the requirements of EMF. Secondly, writing analyses of Smalltalk source code is straightforward using Smalltalk itself, thanks to the high reflectivity of the language [Riv96] (*e.g.,* classes being objects able to provide a list of their methods as objects which can provide their ASTs as objects, which can be traversed to extract data) so this also does not influence the requirements. We only need to provide an abstract interface to all required information about a project being analyzed. Please note that for less reflective languages where source code is not modeled within the language (*e.g.,* Java, C++) we would need to introduce a dedicated way of analyzing projects from their ecosystems. Tools such as Moose [NDG05] or Rascal [HKV12] would be good candidates. To store the results of the analysis, any off-the-shelf database system can be used, and this is the final part of the use case diagram that does not carry requirements for EMF.

Keeping the desired use case activities in mind we can specify a set of functional and non-functional requirements from EMF in the following subsections.

### 3.2.1 Functional Requirements

**FR.1** The system enables a developer to specify actions needed to execute an analysis.

**FR.2** The system provides the developer with an abstract interface to required information about a project being analyzed.

**FR.3** The system provides a centralized database system to store the results of the analysis.

**FR.4** The system executes all the specified analyses on all projects in the ecosystem.

**FR.5** The system provides control over data staleness *i.e.,* provides reasonably fresh data from the ecosystem.

**FR.6** The system provides access to gathered data for the tools that need it.

### 3.2.2 Non-Functional Requirements

**NR.1** Compatibility with the Linux operating system.

## 3.3 EMF Overview

In order to satisfy **FR.5** and **FR.6** we decided to implement EMF using a simple client-server architecture. This means that the server side of the framework is in charge of gathering ecosystem data, storing it and providing it to clients *i.e.,* developer tools. A high level overview of EMF is shown in Figure 3.2. We can see the clear distinction between the server side components and the clients. The clients gather data through the data providing module, which reads from the database. For the database module we use MongoDB[1]. We chose MongoDB for its schemaless data storage model which enables faster development and experimentation. The data providing module is an existing implementation of a MongoDB Java REST server[2]. These two modules fulfill **FR.3** and **FR.6** completely.

The remaining module in Figure 3.2 is the data gathering module. This module deals with **FR.1**, **FR.4** and **FR.5**. It has three distinct parts:

**Configuration** is an XML file used to fulfill **FR.1** and is explained in detail in Subsection 3.3.2.

**EMF Analysis Core**[3] is a Smalltalk package used to provide the developer with a way to interact with various aspects of the framework from Smalltalk *e.g.,* obtaining the database connection, obtaining all classes from the project being analyzed *etc.* This is put in place to satisfy **FR.2** and is further detailed in Subsection 3.3.3.

[1] `https://www.mongodb.org/`
[2] `https://sites.google.com/site/mongodbjavarestserver/`

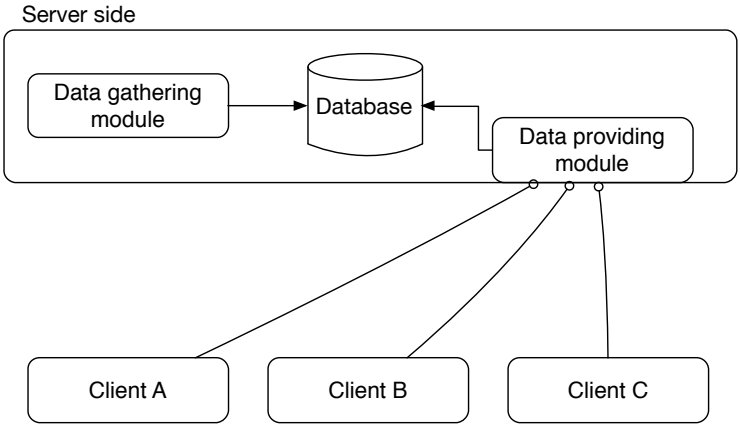


Figure 3.2: High level overview of the components of EMF.

**Execution Engine** is the backbone of the framework. It is the implementation of the execution of analyses defined in the configuration file across the ecosystem projects. To satisfy **NR.1** and **FR.5** we implemented this part as a set of shell scripts to be run at a fixed interval through the chronjob framework of the Linux operating system. A detailed description can be found in Subsection 3.3.1.

### 3.3.1 Execution Engine

The EMF data gathering module[4] is implemented as a set of shell scripts to be run as at a fixed interval through the chronjob framework of the Linux operating system. This makes the EMF portable across any Linux operating system supporting shell and the chronjob framework. It also makes extension, porting and modification very flexible and easy by editing the scripts.

The entry point for the scripts is the deployment script (*deploy.sh*). This script is called from the cronjob and it deploys the entire framework by copying the other scripts to a safe execution directory (in our case */tmp*), executing the next three phases of the

[4] `https://github.com/boris-spas/ecosystemMonitoringFramework`

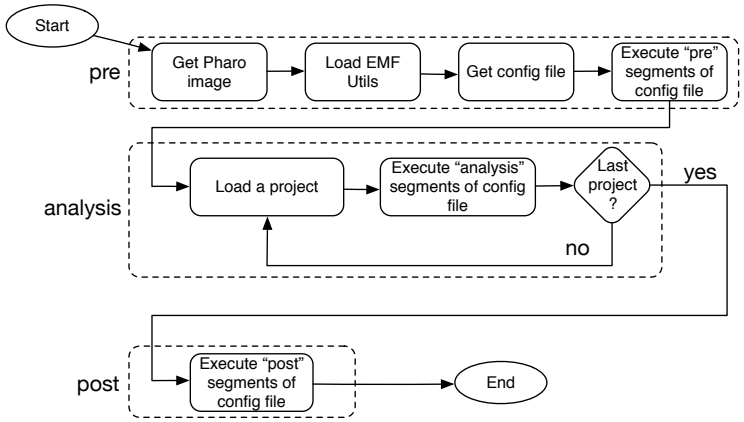


Figure 3.3: An overview of the internal steps of the data gathering module.

EMF (*main.sh*). The next three phases are presented in Figure 3.3 and each phase has a corresponding shell script:

**Preparation Phase** Prepares the stage for executing the analysis. In our implementation this includes obtaining a fresh Pharo image and virtual machine, loading the EMF Analysis Core package and obtaining the configuration file. The configuration file is obtained from a separate repository to enable modification of the file independently from the usage. Once the configuration file is obtained, the script extracts from the configuration file all the source code defined to be executed before running the analysis and executes it.

**Analysis phase** This script is in charge of obtaining each project from the ecosystem and executing each of the analysis source code chunks defined in the configuration file for every ecosystem project. In our case, we use the configuration browser in Pharo as the meta-repository that defines the ecosystem *i.e.,* projects defined in the configuration browser are considered the ecosystem. This script contains two modes of execution: sequential, where projects are processed one by one; and parallel, where we leverage the "simple parallel bash" framework[5]

[5] `https://github.com/boris-spas/simpleParallelBash`

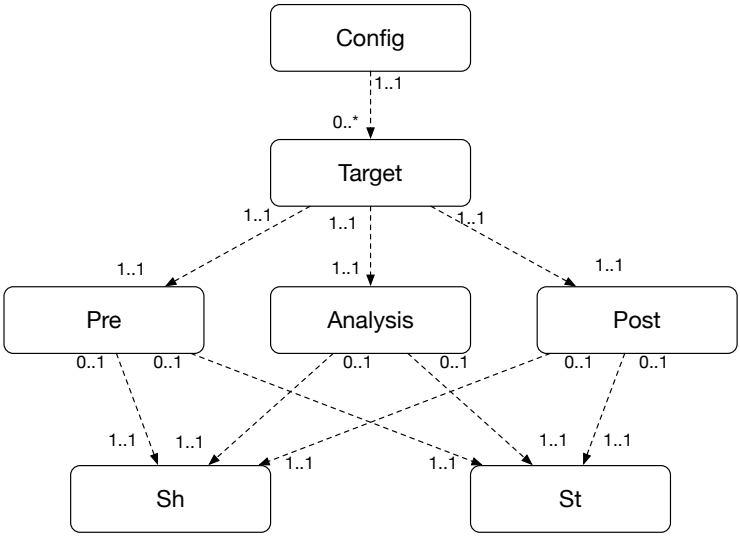


Figure 3.4: The structure of the XML schema for the configuration file represented as a UML diagram.

to process a number of projects in parallel.

**Post phase** Executes all the post analysis source code from the configuration file. This is useful if, for example, the gathered data requires post processing.

### 3.3.2 Configuration

To satisfy **FR.1** we use an XML configuration file. This file allows the developer to specify how to execute an analysis including any steps that need to be performed before and after running the analysis on all ecosystem projects. A graphical representation of the XML schema of the configuration file is given in Figure 3.4.

From the schema we can see that the root element is called "config" and is defined as a sequence of elements called "target". Each target element corresponds to one ecosystem aware tool. It consists of three elements: "pre" – which contains information on steps to be executed before running the analysis (*e.g.,* obtaining analysis source code, setting up a specific collection in the database to store the

analysis results, *etc.*); "analysis" – which contains the steps to be executed in order to run the analysis on a single project (these steps will be repeated for every ecosystem project); and finally "post" which describes any steps that need to be taken after the entire analysis has been completed (*e.g.,* cleanup of temporary resources, post processing the gathered data, *etc.*). Any of these elements have optional "sh" and "st" sub elements that respectively wrap shell commands and Smalltalk code to be executed. A description of an example configuration file can be found in Subsection 3.5.3.

### 3.3.3 EMF Analysis Core

*EMF Analysis Core*[6] is a Smalltalk package used to provide the developer a way to interact with various aspects of the framework from Smalltalk. Its main task is to address **FR.2** and **FR.3** but it also provides some useful features inspired by the needs that arose while developing ecosystem-aware tools using EMF.

The EMF Analysis Core package contains one class: `EMFAnalysisCore`. This class provides all the necessary features via the following class side methods:

`baseClasses` Returns a set of classes that are part of the base Smalltalk image *i.e.,* all the classes present in the image before loading the project to be analyzed.

`projectClasses` Returns a set of classes that are part of the loaded project.

`projectIndex` Returns the index of the loaded project in the configuration browser. This enables differentiation between projects.

`projectName` Returns a `String` object containing the name of the loaded project, as displayed in the configuration browser. This also enables differentiation between projects.

`executionID` Returns a `String` object containing a time stamp when the `EMFAnalysisCore` class was loaded into the image. This enables differentiation between different runs of EMF.

[6] `http://smalltalkhub.com/#!/~spasojev/EMF`

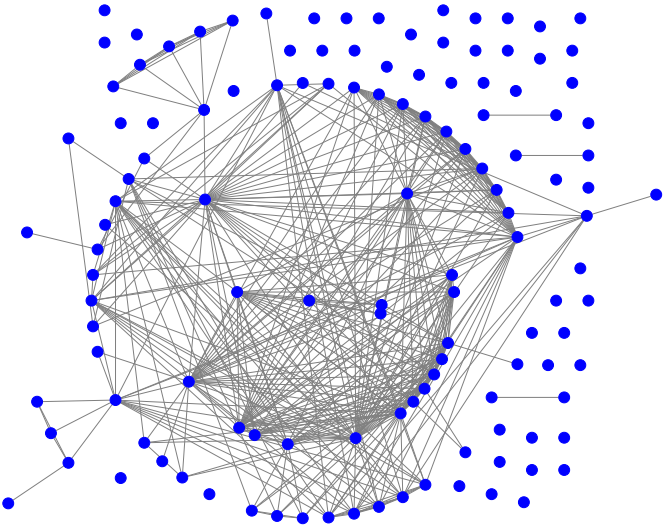

Figure 3.5: Dependencies between projects in the Pharo Configuration browser.

## 3.4 The Ecosystem

In this section we present a short description of the Pharo ecosystem as defined in the configuration browser. The configuration browser reads its list of projects from a human maintained meta repository[7] containing scripts that automatically load projects and their dependencies. Many of these dependencies are also defined in the same meta repository, supporting the claim that these projects co-evolve, and that changes in one project affect others throughout the ecosystem. In Figure 3.5 we can see the dependencies (represented as edges in the graph) between projects (represented as the nodes) from the configuration browser. This graph does not show the dependency to the classes from the base image which all the projects have. It is clear from the dense network of edges in the graph that these projects are interdependent.

[7] `http://smalltalkhub.com/mc/Pharo/MetaRepoForPharo30/main/`

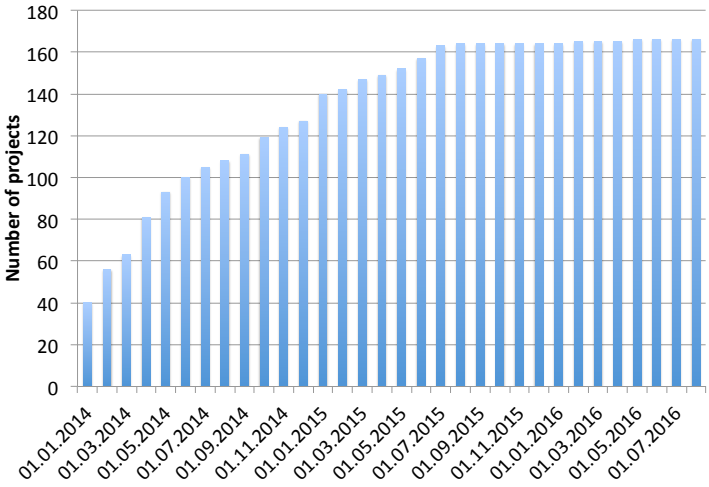


Figure 3.6: The growth of the Pharo ecosystem between 01.01.2014. and 01.08.2016.

To discuss the size of the ecosystem, we present Figure 3.6. In this figure we can see the number of projects in the ecosystem on the first of each month in the interval between 01.01.2014. and 01.08.2016. We can see that there is linear growth of the number of projects until mid 2015, after which the number stabilises around 160 projects. This illustrates that this ecosystem evolves not just in the content of individual projects, but also in its scope, supporting further the need for providing the ecosystem-aware tools with fresh ecosystem data. It also illustrates the advantage of using a meta repository as the source of the ecosystem projects rather than using a fixed set of projects.

## 3.5 Implementing Ecosystem-aware Tools with EMF

In this section we describe the process of implementing a simple ecosystem aware tool. The tool was chosen to be as simple as possible while illustrating the features of EMF and the process of ecosystem-aware tool development.

```
{
  "className" : "CppTranslatedPrimMethod",
  "project"    : "CTranslator"
}
```

Listing 3.1: Example output of the class name clash prevention tool back end in JSON form.

### 3.5.1 Class Name Clash Prevention Tool

Pharo Smalltalk does not have a concept of namespace [GR83]. This means that different projects could define a class with the same name, which could cause problems if such project needed to co-exist in the same image. We call this a class name clash. Class name clash prevention tool[8] [9] is an ecosystem-aware tool that informs a developer if a newly created class shares a name with another class in the ecosystem, thus avoiding later class name clashes. This tool was developed by one developer and took around 5 hours to develop. This supports the claim that EMF enables fast and easy development of ecosystem-aware tools.

All ecosystem-aware tools that leverage EMF are developed in 3 steps: developing the back end, registering the back end with EMF and developing the front end. The following subsections describe each of the steps in general and in the case of the class name clash prevention tool.

### 3.5.2 Back End

The role of the back end part of a ecosystem-aware tool is to gather relevant data from the ecosystem and store it for later access by a front end. In the case of the class name clash prevention tool relevant data are class names from all ecosystem projects associated with the name of the project that defines it.

Leveraging the `projectClasses` and `projectName` feature of `EMFAnalysisCore`, creating such an association is trivial. Since the default database in EMF is MongoDB we store this data as JSON documents represented in Pharo Smalltalk as Dictionary objects.

[8] `http://smalltalkhub.com/#!/~spasojev/ClassClash`
[9] `http://smalltalkhub.com/#!/~spasojev/ClassClashFrontEnd`

```
1  ClassClashDataGatherer>>emfAnalysis
2    | toBeStoredToDbCollection projectName |
3    projectName := EMFAnalysisCore projectName.
4    toBeStoredToDbCollection := OrderedCollection new.
5    EMFAnalysisCore projectClasses
6      do: [ :pc |
7        | classToProject |
8        classToProject := Dictionary new.
9        classToProject at: 'className' put: pc name.
10       classToProject at: 'project' put: projectName.
11       toBeStoredToDbCollection add: classToProject ].
12   EMFAnalysisCore
13     save: toBeStoredToDbCollection
14     toDB: 'classClash'
15     inCollection: 'classClash' , EMFAnalysisCore executionID
```

Listing 3.2: Implementation of the class name clash prevention tool back end.

An example JSON object is shown in Listing 3.1. All the functionality of the class name clash prevention tool back end is wrapped in the `emfAnalysis` method of the `ClassClashDataGatherer` class, the source code of which is shown in Listing 3.2. As with all EMF back end analyses, this analysis is written in the context of one project, but executed on each of the projects in the ecosystem.

On line 3, this method initializes a local variable `projectName` by using the `EMFAnalysisCore`. When this analysis is executed by the framework on each of the ecosystem projects, this variable will be initialized to the name of each of the projects.

On line 4, a collection is crated to store the data for batch writes to the database. Line 5 uses `EMFAnalysisCore` to obtain all the classes of the project being analysed, and lines 6 to 11 iterate over each of the classes creating a `Dictionary` object associating the class with the project, and storing it in the collection for later storage to the database.

Lines 12 to 15 use `EMFAnalysisCore` to store the gathered data to a database in a unique collection identified by the `executionID` of `EMFAnalysisCore`

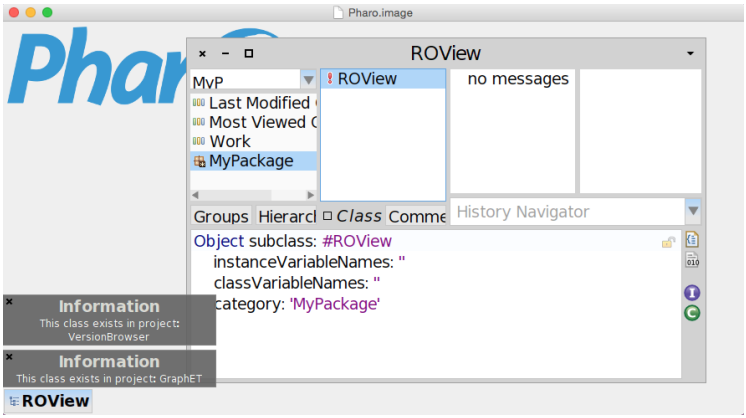


Figure 3.7: The class name clash prevention tool front end notifying a class exists in other projects..

### 3.5.3 Config File

As described in Subsection 3.3.2 EMF uses an XML configuration file to describe the steps needed to perform an analysis on a project. The configuration file for the class name clash prevention tool is shown in Listing 3.3. This configuration contains only one target (lines 2–18) as it is used only for the class name clash prevention tool. This target has a "pre" step (lines 3–12), specifying the Gofer[10] script that loads the source code of the tool's back end described in Subsection 3.5.2 and an "analysis" step (lines 13–17) specifying how to run the loaded back end code. This is done by invoking the `emfAnalysis` method on an instance of `ClassClashDataGatherer` (line 15). Since no post processing is needed for this tool, the "post" step is absent from the configuration.

### 3.5.4 Front End

Once the EMF cronjob runs, and the data is gathered by the class name clash prevention tool back end, it is available for serving by

[10]Gofer is a system for automated loading of source code and dependencies. A description can be found at the following URL: `http://pharobooks.gforge.inria.fr/PharoByExampleTwo-Eng/latest/Gofer.pdf`

```
<config>
  <target>
    <pre>
      <st>
        Gofer new
          url: 'http://smalltalkhub.com/mc/spasojev/
    ClassClash/main/';
          package: 'ConfigurationOfClassClashBackEnd';
          load.
        (Smalltalk at: #ConfigurationOfClassClashBackEnd)
        loadDevelopment
      </st>
    </pre>
    <analysis>
      <st>
        ClassClashDataGatherer new emfAnalysis.
      </st>
    </analysis>
  </target>
</config>
```

Listing 3.3: EMF configuration file for the class name clash prevention tool.

the data providing module of EMF. At this point, the front end of the class name clash prevention tool can function. We implement the front end as a plugin for Nautilus[11] the default system browser for Pharo. This plugin reacts to a new class being created by sending, via a HTTP POST request, the name of the new class to the data providing module which returns from the database a list of all projects that contain a class with that name. This list, if not empty, is then presented to the developer as a series of pop-up notifications as shown in Figure 3.7. In this figure we can see the situation right after a user has created a class called `ROView`. The dark grey information boxes in the bottom left corner of the figure are the pop-up notifications that the newly created class exists in other projects in the ecosystem.

[11] `http://smalltalkhub.com/#!/~Pharo/Nautilus`

## 3.6 Conclusion

A large body of work exists about integrating ecosystem data *i.e.,* data from other related projects, into the development process. Current state of the art in the field relies on custom implementations for data gathering for each individual tool, we observe that the data gathering process can be automated and standardized.

We provide a proof of concept implementation for a unified framework for developing ecosystem-aware tools for Pharo Smalltalk called the Ecosystem Monitoring Framework or EMF. EMF fully automates the process of running analyses on the Pharo ecosystem, storing the resulting data, providing it to tools and keeping it up to date. Unifying all these steps into one framework enables the developer to focus on the data the tool requires and ways to make use of that data rather than on the technical part of loading projects and gathering the data.

This chapter also presents a very simple tool used to prevent class name clashes in Pharo Smalltalk projects. This tool illustrates all the steps needed to develop an ecosystem-aware tool using EMF, and gives insight into the amount of effort needed to develop one such tool.

# 4

# Ecosystem-Aware Type Inference

## 4.1 Introduction

Software developers spend more time on maintaining and evolving existing software than writing new code. Maintenance consumes over 70 percent of the total life-cycle cost of a software product [BB01]. This means that support for reading and understanding code is very important. Static type information in source code helps developers understand how the software system works [MHR+12], but the expressiveness provided by dynamically typed languages can make developers more productive [Han10]. Many attempts have been made at getting the best of both through type inference or optional typing.

Most type inference techniques rely on statically analysing the source code of the software system in question, and using the gathered data to infer the possible types. A basic approach is to track the usage of a variable *i.e.,* the messages sent to it, and infer the type by determining which classes implement the corresponding methods. This can lead to false positives, *i.e.,* types that match the required interface but can never actually be reached at run time. More advanced techniques perform deeper analysis *i.e.,* using data flow or control flow, but ultimately suffer from the same problem of false

positives. A developer faced with a provided set of possible types cannot easily identify the false positives.

This chapter presents an approach to augment existing type inference techniques by supplementing the information available in the source code of a project with data from other projects from the same software ecosystem [Lun09]. By using the data from the ecosystem, it is possible to increase the amount of information used to infer types and thus help avoid and identify potential false positives. For all available projects from the ecosystem, we track how often messages are sent to instances of available types throughout the source code. With this information, we can sort the potential types of a variable the developer cares about based on their likelihood of being the actual type in the context. The likelihood is computed based on how many times the messages sent to this variable have been observed to be sent to each potential type throughout the ecosystem.

We have implemented a proof-of-concept prototype and used it to evaluate the approach. We show that, for our implementation, measuring the frequency of association between a message and a type throughout the ecosystem source code is helpful in identifying correct types. The evaluation data shows a substantial increase in the number of correctly inferred types.

The chapter is organized as follows: Section 4.2 gives a high level overview of the problem and the proposed approach to solving it; Section 4.3 presents the related type inference techniques; Section 4.4 gives a detailed description of the proposed approach, as well as the formal model used to describe it; Section 4.5 presents the details of the implementation of the prototype; Section 4.6 shows the methods and results of the evaluation of the prototype; and finally Section 4.7 concludes and discusses future work.

## 4.2 Overview

To better understand the contributions of this chapter, we take a look at an existing three-step approach to type inference [PMW09]. We start from this approach because it is simple to understand and implement, is reasonably fast and is representative of its field. Other more complex approaches would gather more data about the system to increase precision, and such complexity is unneeded for this purpose.

The approach has three steps:

1. Interface type extraction. This phase reconstructs the type of a variable of interest by using static analysis to find all messages sent to it within the context of the given class. The system is then searched for all classes that implement this set of messages.

2. Assignment type extraction. This phase reconstructs the type with respect to the assignments to the variable. This is a heuristic based analysis of the right side of assignments to the variable in question.

3. Merger. Merging the results from phases one and two into the final type results for the variable. Several different ways exist to do the merge [PMW09], but we focus on the one that gives priority to the assignment type, and moves to interface types if an assignment type does not exist .

This "single-system type inference" (SSTI), is not helpful in cases where the amount of data gathered by the first two phases is limited. To understand this limitation consider the example in Listing 4.1.

The example[1] is written in Pharo Smalltalk. The first part (lines 1–7) declares a new class called `MethodBrowser` and lists the instance variables of the class. Special attention for this example is put on the instance variable `toolbarModel`. The initialization of this variable is not shown in the example for simplicity but is done using the factory design pattern. The second part is the definition of a method named `initializePresenter`. This method is the only place `toolbarModel` is used and the only usage is sending it the message `method:`.

Suppose the developer needs to know the type of the `toolbarModel` instance variable. She could care about this in order to understand which of the implementations of `method:` will be invoked when this code is executed or just use the knowledge of the type of this variable to better understand the entire system. In Smalltalk, good practices recommend instance variable names to match the type of the variable. However, we find no ToolbarModel class in the system.

[1]The code snippet is actual code from the Spec system, to be found at `http://smalltalkhub.com/#!/~Pharo/Spec`

```
ComposableModel subclass: #MethodBrowser
   instanceVariableNames: 'listModel textModel toolbarModel'
   category: 'Spec-Examples-PolyWidgets'

MethodBrowser>>initializePresenter
   listModel whenSelectedItemChanged: [:selection |
      selection
         ifNil: [
            textModel text: ''.
            textModel behavior: nil.
            toolbarModel method: nil ]
         ifNotNil: [:m |
            textModel text: m sourceCode.
            textModel behavior: m methodClass.
            toolbarModel method: m ]].

   self acceptBlock: [:t |
      self listModel selectedItem inspect ].
   self wrapWith: [:item |
      item methodClass name,'>>#', item selector ].
```

Listing 4.1: The type of toolbarModel cannot be detected by the single-system approach

Applying the previously described approach to this instance variable would produce 21 possible classes. This is due to the fact that there are no assignments of an explicit type to the variable, only one method is invoked, and the method in question is defined in all 21 classes. This means that if the developer wishes to understand which implementation of `method:` is invoked, she is in an uncomfortable position of having 21 possibilities. The actual number of possible classes is even larger, because we ignore all subclasses of the classes defining the method in question. Obviously in cases like this, the information provided to the developer is not helpful. Pluquet *et al.* show that, in their evaluation data, on average less than 40% of instance variables receive enough messages and initializations to successfully infer one possible type [PMW09].

An alternative for the developer is to execute the code and observe the run time value (*e.g.,* using a breakpoint). This is not always easy to do, as the system might not be easily executable for a num-

ber of reasons (lack of input to the system, lack of dependencies, long execution time before reaching the position of interest, *etc.*). This is normally countered by observing the execution of unit tests, but the existence of unit tests is not something we can reliably count on.

Given a set of possible types provided by the SSTI it would be helpful to the developer if the set were sorted by how likely each of the types is to be correct. Since the data we have available is a set of selectors[2] that the instance variable receives, we can compare this set to patterns of message sending in other projects from the ecosystem. The intuition is that the more often we find that the same messages are sent to a uniquely identifiable type, the more likely that type is to be correct.

In our case, after the analysis of other systems in the ecosystem we find out that the message `method:` is commonly sent to instances of 3 classes out of the 21 that implement the method. Those are `MethodToolbar`, `ZnRequest`, and `SourceMethodConverter`. The actual type assigned to the instance variable `toolbarModel` at run time is `MethodToolbar`. We argue that type association information from other projects can be beneficial to recovering types.

The proposed approach (Ecosystem-aware type inference, EATI for short) automates the process of using ecosystem data for augmenting SSTI and it consists of two phases. The first is an analysis of a large number of systems that results in the data about the frequency of association of messages and types. The second phase concerns the developer in need of type inference. To infer a type, we attempt SSTI and in case the results are ambiguous we query the data from phase one to receive a collection of possible types sorted by the number of times the messages were sent to types in the ecosystem.

## 4.3 Related Work

EATI addresses *type inference for dynamically typed object-oriented languages* to support program comprehension. The scope of this work is different from classical type inference techniques for statically typed languages like Scala [OAC+06], where type inference

[2]In Smalltalk jargon, a *selector* is the name of a message, *i.e.,* `+` or `method:`, used to select the *method* to respond to the message.

frees the developer from having to specify types that can be inferred.

Much of modern type system research is based on the work of Milner who published the description of a polymorphic type-inference algorithm called "Algorithm W" [Mil78]. It is a fast algorithm, performing type inference in almost linear time with respect to the size of the source code and was first implemented as part of the type system of the programming language ML.

A well known type inference algorithm is the Cartesian Product Algorithm [Age95] (CPA). This algorithm infers concrete types to support performance of parametric polymorphism. CPA does this by partitioning the calling contexts of a method based on the types of the actual arguments passed to the method. It supports dynamically typed languages, as it is implemented originally for Self, and its contribution is limited to inferring concrete types from polymorphic types. CPA is the basis for other type inference engines for other languages *i.e.,* Starkiller [Sal04], a type inferencer and compiler for Python.

A fast type inference technique presented by Pluquet [PMW09] is used as a basis for the prototype implementation of the type inference presented in this chapter. This technique is outlined in the motivating example section. This technique is also used by Milojković *et al.* as a basis for exploring cheap heuristics for type inference based on information only from the system in question [MN16].

The approaches so far use different kinds and quantities of data obtained through statical analysis to infer types. None of them expand to more than one system, so any of them can benefit from EATI. An implementation of EATI on top of these approaches would need to be significantly different from the one presented in this chapter, as it would have to manage the different data used.

Other approaches use the execution of a program to gather types. One such approach is presented by Jong-hoon *et al.* for the language Ruby [DGGH11]. This approach uses wrappers for variables that generate constraints during execution which are later used to infer types. The inferred types can be used for documentation, and thus better code comprehension, but all dynamic approaches are limited by the requirement that the code needs to be runnable, either through test cases, examples or actual execution. Milojković *et al.* use class usage frequency from inline caches of previous executions of unrelated code to improve type inference without the need for

running the program [MBGN16].

Another field of related work has to do with optional typing. Optional typing attempts to enable developers all the benefits of using dynamically typed languages, with the option of specifying types when and where they deem appropriate [ACF+13]. This enables compile-time type checking for provided types, and also enriches the source code with static types. Examples of optionally typed languages are Strongtalk and Gradualtalk — dialects of Smalltalk, Dart — a language developed by Google, and Typescript — an optionally typed Javascript developed by Microsoft. A type inference engine for these languages could be used to infer and generate the optional types. This would free the developer from specifying inferable types, as with Scala. Such engines could also benefit from ecosystem data.

## 4.4 Ecosystem-Aware Type Inference

To explain EATI we introduce a simple set-theoretic model in Figure 4.1 that captures key properties for the entities shown in the UML diagram in Figure 4.2. For simplicity we ignore temporary variables and method arguments throughout the chapter. We can greatly simplify the model and implementation by ignoring them, and, since the approach works the same for these variables, application of the approach to them is trivial.

### 4.4.1 Core Model

Given all the source code in a software ecosystem, $C$ is the set of all classes, $F$ the set of all instance variables (*i.e.,* fields), $M$ the set of all methods, and $S$ the set of all "selectors" (*i.e.,* method names).

A given field $f$ is uniquely defined in a class $c = def_f(f)$ (Equation 4.1). Similarly, each method $m$ is defined in a unique class $c = def_m(m)$ (Equation 4.2). Every class $c$ other than `Object` has a unique superclass $c' = sup(c)$ (Equation 4.3). *sup* is a partial function, since $sup(\texttt{Object}) = \bot$.

Every method $m$ has a unique method name (*i.e.,* a *selector* used to select $m$) $s = sel(m)$ (Equation 4.4), and every method $m$ sends a set of message selectors to a given field $f$, namely $sends(m, f)$ (Equation 4.4).

$$def_f : F \to C \tag{4.1}$$

$$def_m : M \to C \tag{4.2}$$

$$sup : C \to C \tag{4.3}$$

$$sel : M \to S \tag{4.4}$$

$$sends : M \times F \to 2^S \tag{4.5}$$

$$def_f(f) \notin sup^*(def_m(m)) \Rightarrow sends(m, f) = \emptyset \tag{4.6}$$

Figure 4.1: The core model. $F$ = fields, $C$ = classes, $M$ = methods, $S$ = selectors.

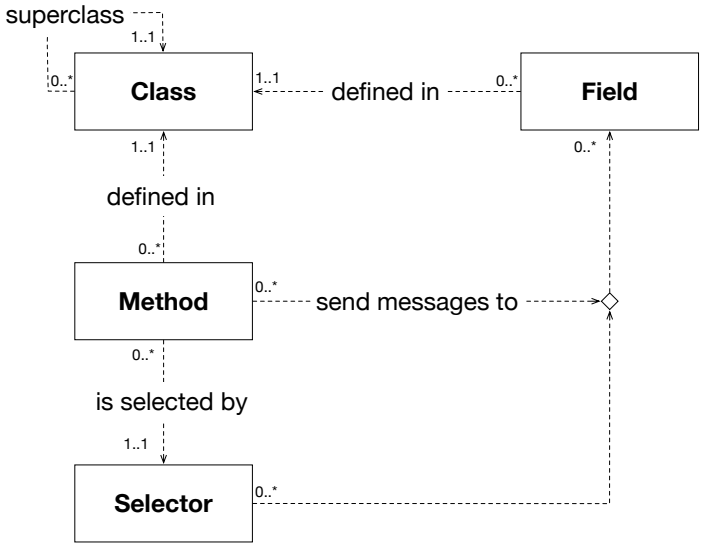


Figure 4.2: The core model in UML.

Note that a method $m$ can only access fields defined in the same class where $m$ is defined, or fields inherited from one of its superclasses. For all other combinations of $m$ and $f$, $sends(m, f)$ returns the empty set (Equation 4.6).

Consider the example class hierarchy Figure 4.3. In the example, $def_f(\texttt{rect}) = def_f(\texttt{tri}) = def_m(\texttt{main}) = \texttt{DrawEditor}$, as both these

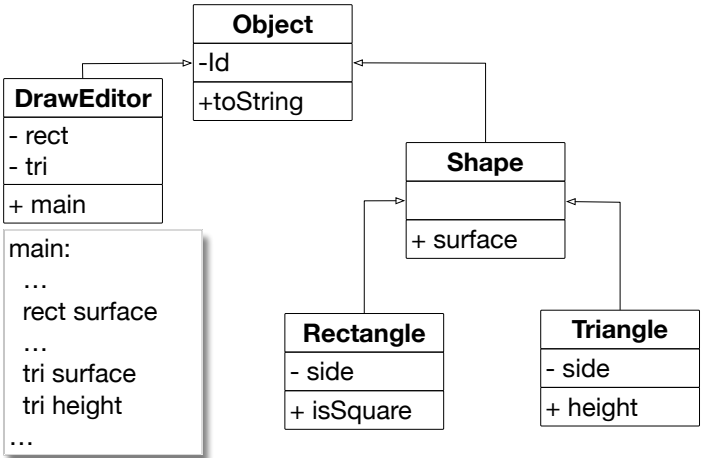


Figure 4.3: Sample class hierarchy with the implementation of one method .

instance variables and the method `main` are defined in the class `DrawEditor`. Also, $sup(\texttt{DrawEditor}) = sup(\texttt{Shape}) = \texttt{Object}$ which is obvious from the hierarchy.

Note that in the example we are implicitly equating fields and methods with their name, even though multiple fields or methods could have the same name. This is done for simplicity and will be done in all further examples in this section.

We can now query the model to compute metrics that will allow us to rank the results of type inference, as summarized in Figure 4.4. The *interface* of a class $c$, $ifc(c)$, is the set[3] of selectors of all methods defined in $c$ and its superclasses (Equation 4.7). The selectors *received* by a field $f$, $rec(f)$ is the set of all selectors of messages sent to $f$ by all methods defined in the same class $c = def_f(f)$ (Equation 4.8)[4].

Returning to the example, the interface of the class `Rectangle` is:

$$ifc(\texttt{Rectangle}) = \{\texttt{isSquare}, \texttt{surface}, \texttt{toString}\}$$

[3]Note that we implicitly extend $sel$, $def_m$ and other functions in the usual way to take sets of values as arguments and similarly return sets of values.

[4]This definition is consistent with classical SSTI [PMW09]. Later we will consider messages sent by methods in subclasses as well.

$$ifc(c) = sel(def_m^{-1}(sup^*(c))) \tag{4.7}$$

$$rec(f) = sends(def_m^{-1}(def_f(f)), f) \tag{4.8}$$

$$types(f) = \{c \in C | rec(f) \subseteq ifc(c)\} \tag{4.9}$$

$$roots(C') = \{c \in C' | \forall n > 0, sup^n(c) \notin C'\} \tag{4.10}$$

$$unique(f, c) = \begin{cases} 1 & roots(types(f)) = \{c\} \\ 0 & o/w \end{cases} \tag{4.11}$$

$$selscore(c, s) = \sum_{f \in F, s \in rec(f)} unique(f, c) \tag{4.12}$$

$$classscore(c, f) = \sum_{s \in rec(f)} selscore(c, s) \tag{4.13}$$

Figure 4.4: Computing class scores over the core model.

as those are all the methods defined in it and its superclasses. Looking at the implementation of method `main` we can see that messages are sent to the instance variables `rect` and `tri`. We can express which messages with

$$rec(\texttt{rect}) = \{\texttt{surface}\}$$

and

$$rec(\texttt{tri}) = \{\texttt{surface}, \texttt{height}\}$$

The set of possible *types* of a field $f$, $types(f)$, is the set of classes whose interface includes all selectors received by $f$ (Equation 4.9). The roots of the hierarchies of a set of classes $C'$ is the subset of those classes without superclasses in $C'$ (Equation 4.10).

Applying this to our example we get

$$types(\texttt{rect}) = \{\texttt{Shape}, \texttt{Rectangle}, \texttt{Triangle}\}$$

and

$$types(\texttt{tri}) = \{\texttt{Triangle}\}$$

A field $f$ is inferred to be of a *unique* type $c$ if the set of inferred types for $f$ has a unique root $c$. The function $unique(f, c)$ returns

a count of 1 for all such fields (Equation 4.11). This means that $unique(f, c)$ will, in our example, equal 1 in only two cases.

$$unique(\texttt{rect}, \texttt{Shape}) = 1$$

$$unique(\texttt{tri}, \texttt{Triangle}) = 1$$

Now we compute the *selector score* of a class $c$ with respect to a selector $s$, $selscore(c, s)$, as the number of fields $f$ that are determined to be of the unique type $c$, where $s$ is sent to $f$ (Equation 4.12). A few selector score values for combinations of classes and selectors from our example are

$$selscore(\texttt{Shape}, \texttt{surface}) = 1$$

$$selscore(\texttt{Rectangle}, \texttt{surface}) = 0$$

$$selscore(\texttt{Triangle}, \texttt{height}) = 1$$

Finally, we compute the *class score* of a given class $c$ with respect to a field $f$, $classscore(c, f)$, as the sum of all its selector scores for selectors of messages sent to $f$ (Equation 4.13). The usage of class score is beyond the scope of our small example, but will be explained further in the chapter.

### 4.4.2 Storing Data from the Ecosystem

As with any ecosystem-aware tool, the first step to the proposed approach is to gather type information from projects in the ecosystem and information on message sending to instances of those types.

An entry in the stored data consists of a class name $c \in C$, selectors $\{s_1...s_n\} \subseteq S$ sent to instances of $c$, and the number of times each $s_i \in \{s_1...s_n\}$ has been sent to instances $c$. This number is called the selector score of a class $(selscore(c, s)|s \in S, c \in C)$, as messages with the same selector can be sent to instances of different classes, yielding different selector scores. A sample database is presented in Table 4.1. We aim to show that this information is sufficient for improving the type inference.

The data needed for EATI can be gathered through dynamic or static analysis. In this context, dynamic analysis means gathering

type information from a running system [DGGH11]. This provides actual run-time types, but requires the system to be runnable and produces false negatives — a variable may not actually be bound to a type during the observed execution of the system, but might during others. Static analysis means running a type inference engine on the source code. As we saw in the example from the beginning of Section 4.2, static analysis can often produce false positives — types deemed "possible" that never occur at run time. Since the ecosystem data is large, we can ignore the false positives from running static analysis and avoid false negatives by not doing dynamic analysis.

### 4.4.3 Using the Stored Data

In order to infer types, we apply SSTI, and in case there is more than one possible type, the data gathered from the ecosystem is queried for more information in order to sort the possible types and present the developer with the more likely candidates. The repository query for an instance variable $f \in F'$ should contain $rec(f)$, where $F'$ is the set of all instance variables in the system being typed by the user. The result of the query is a set of possible classes $\{c_1...c_n\} \subseteq C$, determined by which classes in the repository have records of their instances receiving selectors from $rec(f)$. The result of the query should be sorted by the class score.

For example, given the repository table from Table 4.1, querying the repository with a set of selectors {`substring:`, `startsWith:`} would return just the class `ByteString` with a score of 23. `ByteString` is the only result because it is the only class found in the repository whose instances receive the given selectors. The score is due to the selector `substring:` having a score of 9 and the selector `startsWith:` having a score of 14, yielding 23.

If the query contained only the selector `+`, the result would have been a set of classes containing the class `Integer` with a score of 27 and the class `ByteString` with a score of 10. Instances of both classes have been observed to receive this selector, but instances of class `Integer` receive it more often.

## 4.5 Implementation

We have implemented a prototype of EATI using the Ecosystem Monitoring Framework for Pharo Smalltalk. As with other

| Class Name | Selector | Score |
|---|---|---|
| ByteString | substring: | 9 |
| | toUpperCase | 6 |
| | + | 10 |
| | startsWith: | 14 |
| | endsWith: | 4 |
| Integer | + | 27 |
| | − | 63 |
| | toString | 40 |
| | bitAnd: | 2 |

Table 4.1: A sample repository containing information on the frequency of sending certain selectors to classes `ByteString` and `Integer`

ecosystem-aware tools, the implementation consists of three distinct subsystems: a data gatherer analysis that mines usage data from the ecosystem, a GUI available to the developer, and the persistent store through which the other two subsystems communicate.

### 4.5.1 Data Gathering

The data gatherer analysis is tasked with populating a database with information on type-selector relationships from the ecosystem. It attempts to infer the type of all instance variables in the ecosystem and, if successful, stores the relation between the inferred type and the selectors sent to the variable. This is the back end of the ecosystem-aware tool.

The type inference engine used is an improved implementation of the SSTI described in the example from the beginning of Section 4.2 that takes into account field usage data from the subclasses. This provides more information for instance variables that are used predominantly in the subclasses of the class that declares it and requires a small modification to our model redefining $rec$ as

$$rec(f) = sends(M, f)$$

for all $f \in F$. Note that this definition will give empty results for all $m$ where $f$ is not accessible, so only methods of the class $c = def_f(f)$

and its subclasses actually contribute selectors to the result.

The prototype iteratively pre-computes $selscore(c, s)$ for all classes and selectors, starting with the class `Object`, the root of the class hierarchy. Performing a depth-first search of the class hierarchy ensures that all the data from the subclasses of each class are gathered. Once the entire subtree for a particular class has been traversed, the search for adequate types of the instance variables of the class begins, based on the data collected from the subtree.

### 4.5.2 The Store

The pre-computed *selscore* values are represented as a set of triplets:

$$(c, s, selscore(c, s))|c \in C, s \in S$$

and stored as a key-value JSON document. We group all the triplets with the same $c$, and use $c$ as the key for that entry in the database. A textual representation of a sample JSON document follows.

```
{
  "_id" : ObjectId("51b0df6b44b1392c8e7ee0ec"),
  "className" : "Mutex",
  "selectors" :
  {
    "critical:" : 2,
    "ifNil:" : 1
  }
}
```

By the end of the analysis of all the projects the database contains the global score for every available class-selector combination in EMF's central database.

### 4.5.3 The Client

The client is the front end of the entire system, and is the bridge between the user and the system. We implemented a very simple GUI that offers the developer the choice which class should be processed, and whether or not to include its subclasses in the analysis. Processing the class consists of running SSTI on its instance variables, consulting the database when necessary and presenting the results to the developer. There are many ways to use the provided data both by tools and by the developer, but the integration of type

inference results into the development environment is beyond the scope of this thesis.

## 4.6 Evaluation

To evaluate the prototype implementation we populate the store with data from 74 open source projects from the Pharo Smalltalk ecosystem as defined in the Pharo configuration browser.

A total of 8374 classes were analyzed. This produced 746 entries in the store. This means that running type inference on all the instance variables of the classes produced 746 classes $\{c_1 ... c_{746}\} \subset C$ such that

$$\exists f \in F | unique(f, c) = 1, rec(f) \neq \emptyset$$

After populating the store with data gathered from the projects, we take 97 instance variables from 5 projects. These projects are not a part of the set used to populate the store and were chosen because they have unit tests available. We were limited to these 5 projects because of the relatively small size of the Pharo ecosystem and there are few projects with high unit test coverage.

The run-time types of the instance variables are recovered by instrumenting the source code of the projects to log types of objects assigned to instance variables and running the unit tests. These types are held to be the actual types, implications of which are described in the threats to validity section.

Types of these instance variables are then inferred using SSTI and EATI.

It should be noted that the projects used for testing contain a total of 402 instance variables, but most of them were ignored for one of two reasons:

1. A total of 107 instance variables received no messages thus the instance variable is not a valid candidate for this type inference technique;

2. Running the unit tests did not provide a run-time type for 198 instance variables. This is most likely due to poor test coverage.

Throughout the evaluation we try to answer the following questions: How well does SSTI work, what is the improvement with EATI, and when and why does EATI fail? The evaluation discussion is divided into 3 parts, one focusing on discussing successfully inferred types, one focusing on the false positives and the last commenting on the remaining situations in which the single-system approach failed and no data was provided by the ecosystem-aware approach. A summary of the evaluation results is given in Table 4.2.

| Total | Successful | | False positives | | No data |
|---|---|---|---|---|---|
| | Single system | Eco-aware | Selectors only from *Object* | True failures | |
| 97 | 21 | 20 | 23 | 25 | 8 |

Table 4.2: Summary of the evaluation results

### 4.6.1 Successful Attempts

EATI provides a sorted list of the most likely types of an instance variable. In the best case scenario the correct type should be at the top of the list. We declare two scenarios for a successful attempt:

1. if the SSTI infers the correct type (as provided by running the unit tests)
2. if the SSTI provides several types and the correct type is at the top of the sorted list provided by EATI.

The results show 21 instance variable types that have been successfully inferred using SSTI and an additional 20 using EATI. This means that using the EATI almost doubled the number of successfully inferred types.

An analysis of the inferred types shows that EATI works reliably for types from the standard library, as 19 of the 20 instance variables have run-time types from the standard library. These include *Array*, *SmallInteger*, *ByteString* and others. We argue the approach works well with the standard library because it is so widely used, and

the data on type-selector relations is abundant. We expect that the success would generalize to any types sufficiently popular in the ecosystem.

### 4.6.2 False Positives

For an inference to be considered a false positive the types provided by the EATI should be different than the actual run-time type of the instance variable (as provided by running the unit tests). This situation can be more damaging to the comprehension of the source code than not receiving any result at all. This is because it may lead the developer to make wrong decisions based on the wrong type of a variable.

The results show 48 false positives. The number seems unacceptably high, but a second look at the data reveals that almost half of the false positives were caused by the lack of selectors sent to those instance variables. A total of 23 instance variables that caused a false positive received only selectors declared in the `Object` class, such as the `ifNil:` selector used to check if the object in question is a `nil` object[5]. Selectors declared in the `Object` class, can be sent to any Smalltalk object. Thus, those selectors carry no useful type information. We argue that those false positives can be ignored, as they would very easily be identified by checking if all the given selectors are defined in `Object`, which can be easily automated.

Out of the remaining 25 false positives, 6 fail to give the correct type at the top of the list, but the correct type is present in the top three. We call these "near misses".

The remaining 19 false positives each fall into one of two categories:

1. Run-time types not present in the ecosystem — These types are specific to the project in question, and as such are not present in the store *i.e.,* classes only used within this project. Since EATI cannot access source code that uses these classes, they can only be identified through SSTI.

2. Domain specific selectors and types — this false positive arises when selectors used widely for one purpose are used in a different or domain specific manner. For example, the comma

[5] `nil` is an object in Smalltalk. It is the sole instance of the `UndefinedObject` class.

selector (",") when sent to an object of type `ByteString` is used to concatenate strings. On the other hand, in the Petit-Parser [KLR+13] framework this selector is used to create a sequence of parser combinators. Since concatenation of strings is far more frequent than parser combinator sequences, the data in the store suggests that the type of a variable receiving only the selector `,` is a `ByteString`. At this point, it is left to the developer to use her knowledge of the specifics of the project in question to detect this kind of false positive. In future work we would like to explore whether additional context information can be exploited to determine the domain of the project, and adjust the EATI accordingly.

### 4.6.3 No Data from the EATI

A total of 8 instance variables were completely unidentifiable. The selectors sent to these variables are declared in more than one class in the system, thus SSTI results in many false positives. The combination of these selectors has never been linked to a type in the store, *i.e.,* :

$$\nexists c \in C | unique(\texttt{f}, \texttt{c}) = 1$$

This does not mean that all individual selectors from $rec(\text{f})$ have never been seen in the ecosystem. It means that during the data gathering phase no uniquely inferred type has received this combination of selectors.

These situations leave the developer with no insight into the type of a variable, but are also not damaging as they do not mislead like the false positives do.

### 4.6.4 Threats to Validity

Even though EATI is applicable to any dynamically typed language we cannot guarantee the evaluation will generalize to other ecosystems. Although the approach should benefit type inference in any ecosystem, we cannot state in what way without more insight.

Another threat is the fact that unit tests are used to determine the run-time types for instance variables. Unit tests do not provide a complete picture of the running system, and their execution provides a limited set of possible types for instance variables. With that in

mind, the effort to precision ratio for using unit tests is high, and the alternatives of symbolic execution or manually running the systems are significantly more difficult, and are also prone to false negatives. It is an open question to determine whether a given set of unit tests offers a representative picture of the actual types that would be bound during typical (non-test) runs.

It is possible that the selection of SSTI we made could have affected the results. Even though the SSTI we used is representative it is hard to say what effect using other techniques to enable EATI would have on the results. We chose this SSTI for its simplicity and speed, and as such it served the needs of this evaluation. Since EATI is meant as a supplement for SSTI, changing the SSTI engine could increase the SSTI success rate and make EATI seem less potent. On the other hand, if another SSTI were used in the data gathering phase of EATI, the results might normalize. In future work we plan to explore the impact of EATI on other SSTI approaches.

Throughout the chapter we ignore how the fact that different versions of APIs coexisting in the ecosystem could affect the results. The magnitude and implications of this escape the scope of the thesis, but such problems could be addressed by treating different versions of classes in the ecosystem as different classes. This would result in diluting the amount of gathered data per class, impacting results for versioned classes. An alternative approach could be keeping explicit track of the differences between versions and annotating the differences, while keeping the common parts as standard methods of the class.

Finally, using different type inference engines in the data gathering phase and on the client side could yield very different results.

## 4.7 Conclusion and Future Work

This chapter presents a novel approach to type inference that supplements the information available in the source code of a project with data from other projects written in the same language. The approach requires a large set of projects from the same ecosystem to be analyzed, statically or dynamically and indexes the times selectors are associated with different types. After a gathering phase, we store the findings, and a client that infers more than one possible type for a variable can consult the data to sort the candidates and

identify the most likely ones.

The prototype implementation, written for the Pharo Smalltalk ecosystem, enables an analysis of the pros and cons of the approach. The approach has shown to be particularly useful in inferring standard types (`SmallInteger`, `Boolean`, `ByteString`, *etc.*), and the collection types (`Set`, `Dictionary`, *etc.*). We conclude that this is due to the massive usage of these types throughout the ecosystem, and hypothesize that the approach applies to any sufficiently popular type.

In the situations where the approach fails, three patterns can be identified.

Firstly, the situations where no relevant selectors are sent to the instance variable. In these cases the approach will offer either no solution or an unhelpful one (*i.e.,* `Object` class). We conclude that these cases are easily identifiable by presenting to the developer not just the type recommendations but also the set of selectors sent to the instance variables in question. If the selectors are too few or too generic (*i.e.,* the ones defined in the `Object` class) the developer or a tool will be able to conclude that the recommendations are based on insufficient data.

Secondly, the approach does not work on types that are specific to the project in question. For all the types used only in the project, the ecosystem data is useless as it is completely oblivious to the existence of these types. If these types cannot be inferred by the data available in the project that defines them, they receive no benefit from ecosystem data either. Such cases can be easily recognized.

Finally, in situations where selectors are commonly used for operations on one type, but less commonly for other operations on different types, the approach will favor the more frequent one in all situations. We conclude that these cases are not faulty, as the EATI generates recommendations solely based on the highest count of messages sent to types. Once again, it is up to the developer to be aware of the domain of the project and the issues that may arise from it. As future work, we plan to explore how knowledge of the project domain can be used to automatically adjust the rankings.

Throughout the analysis of the approach described in this chapter, several opportunities for improvement have arisen. With more engineering effort it would be possible to solve some of the shortcomings, *i.e.,* allowing the developer to specify the domain or by attempting to infer the domain automatically.

# 5

# Breaking Alphabetical Ordering

## 5.1 Introduction

Alphabetical organization of items can be found in both paper-based telephone books and API documentation systems. While its merits are incontestable in the former, we argue that it is superfluous in the latter and we propose that it be replaced with alternatives that are informed by actual developer needs.

As a testimony to the success of code reuse, an average project will have several dependencies to source code written by third parties [LPS11]. However, reuse also comes with challenges, one of the main ones being learning a new API [Sca06]. Since browsing the source code of upstream dependencies is often not feasible, API documentation is generated automatically from the source code to present synthetic details of entities and their behavior.

Mainstream documentation browsers and code browsers present the methods of an API in alphabetical order. We assume that the main reason for the existence of such an ordering is the fact that it is easy to implement rather than that it is easy to use. Indeed, there are two use cases in which a developer needs to refer to an API documentation page:

- When looking up details for a known method.
- When finding the name for a given functionality that he knows should exist.

In neither case does alphabetical ordering help. In the first case, search is faster than scrolling and visually hunting the right artefact. In the second case, alphabetical ordering is as good as any arbitrary ordering since it does not in any way increase the likelihood of the desired functionality being found.

We aim to improve the way API documentation is presented to a developer by obtaining the frequency of use of all the API methods, and listing the most commonly used ones first, in the case where such usage information can be obtained.

To motivate our approach we mine the frequency of use of API methods from the source code of a large number of projects. Through this analysis we obtain information on all call sites in the source code — which method is invoked[1] on an instance of which class. This data encapsulates the frequency of use of each encountered method, as the number of invocations directly indicates the popularity of the method. In Section 5.2 we present more details.

By obtaining the data this way we ignore where the method is declared, and focus on where it is used. This means that inherited methods are treated the same as declared methods.

We conduct an analysis of the data gathered from the corpora in order to conclude if presenting a small number of commonly used methods first would be beneficial. The aim of this analysis is to answer two questions:

1. How are invocations of methods of a class distributed?
2. How well does alphabetical sorting reflect the frequency-of-use data?

We find that typically 60% of the invocations of methods of a given class are to just 10% of its methods. This shows that a small set of methods is typically very popular compared to the others, resulting in a strongly skewed distribution of method popularity. This is in line with similar studies of software metric distributions [VLBN09]. Details on this analysis and its results are given in Section 5.3.

[1] We use the term "method invocation" to refer to a call site in the source code, not a run-time invocation

Our data show that alphabetical sorting of methods is in no way better than sorting methods randomly, in the context of frequency of use. We calculated the average distance between the index of a method when sorted according to frequency of use and the index when sorted alphabetically. We also made the calculation with the index of a method in a randomly sorted set, and the results differ insignificantly. Details on this analysis and its results is given in Subsection 5.3.2.

Assuming that frequently used methods are frequently searched for, we propose that documentation browsers should augment documentation with information on which methods are more frequently used than others. In Section 5.4 we describe our proposed solution, as well as give an overview of a proof-of-concept implementation and its small initial evaluation. In Section 5.7 we conclude.

## 5.2 Experimental Setup

To obtain the frequency of use of API methods we analyzed the source code of a large corpus of software systems. We ran our ecosystem analysis on the QualitasCorpus [TAD+10] version 20120401r, which contains 112 systems written in Java[2]. We use QualitasCorpus as a snapshot of a software ecosystem because all the projects in the QualitasCorpus share dependencies towards a set of libraries, and some depend on other projects from the QualitasCorpus.

To run the analysis we used Pangea[3], a tool for running language independent analyses on corpora of object-oriented software projects.

The result of our analysis is a set of triplets

$$(c, m, n) \tag{5.1}$$

stored in a database. A triplet signifies the following: in all analyzed projects, we found $n$ call sites where the method $m$ was invoked on an instance of class $c$. Note that, since we are only doing static analysis of the projects, $c$ is the declared type of the variable rather then the actual run-time type which could be different (*i.e.,* a sublclass of $c$) due to polymorphism.

[2]Our analysis infrastructure could not handle one of the systems in the corpus

[3]`http://scg.unibe.ch/research/pangea` — All URLs verified in June 2014.

The triplets can be grouped by a given class $c$, which means that the number $n$, associated with method $m$ summarizes the frequency of use of that method in the context of class $c$. The total number of classes found is 101844.

## 5.3 Analysis

We introduce a simple model of the conducted ecosystem analysis in Figure 5.1.

Given all the source code in a software ecosystem, $C$ is the set of all used classes, $M$ the set of all methods, and $I$ the set of all call sites in the source code. Each method is defined in one class. This is described by Equation 5.2. Note that $M$ is the set of all methods actually invoked throughout the ecosystem — classes potentially define other methods that are not used. Equation 5.3 and Equation 5.4 state that on every call site only one method can be invoked on an instance of one class. Equation 5.5 is an inverse of Equation 5.2 returning all methods of a given class and Equation 5.6 and Equation 5.7 return a set of call sites for a given class or method.

$$def : M \to C \tag{5.2}$$

$$cs_c : I \to C \tag{5.3}$$

$$cs_m : I \to M \tag{5.4}$$

$$methods(c) = def^{-1}(c) \tag{5.5}$$

$$sites(c) = \{i \in I | cs_c(i) = c\} \tag{5.6}$$

$$sites(m) = \{i \in I | cs_m(i) = m\} \tag{5.7}$$

Figure 5.1: The core model. $C$ = classes, $M$ = methods, $I$ = call sites. The *methods* function returns a set of methods defined in a class, and *sites* functions return call sites related with the argument.

Since we are interested in observing API classes, we define $C'$ in Figure 5.2. This is a subset of all the classes that have more than 1000 call sites and more than 10 methods invoked on their instances. The "1000 call sites" criterion is an arbitrary cutoff point that filters out all the classes that are not popular enough to be

$$C' \subseteq C \tag{5.8}$$

$$\forall c \in C', |sites(c)| > 1000, |methods(c)| > 10 \tag{5.9}$$

Figure 5.2: Definition of $C'$ — the subset of classes with more than 10 methods used and the highest number of method invocations.

considered API classes, and the "10 methods invoked" filter removes classes with too few methods invoked, as they are not representative for creating the distribution. The number of classes in $C'$ is 342 and manual inspection shows that they are API classes — classes used in several different projects.

### 5.3.1 Method Call Distribution

In this subsection we aim to answer the question: What percentage of all method invocations of a class constitutes the most popular 10% of methods? Note that we are only looking at invoked methods of a class, so our results are optimistic, meaning that the resulting percentage can only be higher if the class declares additional, unused methods.

The motivation for this framing of the question comes from a manual inspection of usage data for several popular API classes. We noticed as a recurring pattern that only a small number of methods are amongst those most frequently invoked. An example of the distribution for the `java.lang.Thread` class is shown in Figure 5.3. All of the classes we analysed manually exhibit similar distributions.

For all classes in $C'$ we calculate the *invocation inequality grade* of the class. The invocation inequality grade is defined in Equation 5.10 where *topTenPercent* of a class is the number invocations of the most popular ten percent of methods. It is essentially the ratio of the number of times the most popular top ten percent of methods were invoked and the total number of call sites, expressed as a percentage.

Figure 5.5 shows the box plot of the invocation inequality grade for all classes in $C'$. The median is 59.64% (59.04% average). The first and third quartile are 46.54% and 70.96% respectively. From this distribution we can conclude that classes in $C'$ tend to have a

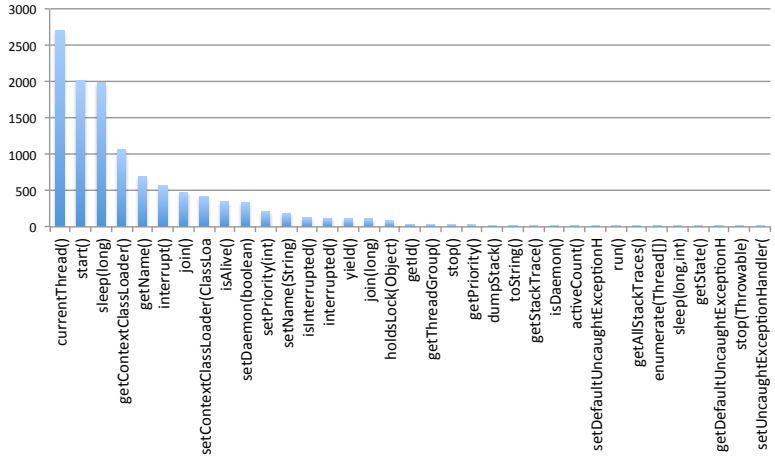


Figure 5.3: Call site distribution per method of class *java.lang.Thread*. The horizontal axis shows method names and the vertical axis shows the number of times the method was invoked.

$$ii\_grade(c) = \frac{topTenPercent(c, 1)}{|sites(c)|}[\%] \tag{5.10}$$

Figure 5.4: Definition of the invocation inequality grade of a class $c$. The grade represents the ratio of the number invocations of the most popular ten percent of methods and the total number of call sites, expressed as a percentage

small number of methods invoked frequently.

### 5.3.2 On Alphabetic Sorting

We argue that sorting methods by frequency of use is a better way to highlight the more important methods. If we assume this to be true, the question is how different is alphabetical sorting when compared to "frequency of use" sorting?

To answer this question we ran another analysis on our data. We analyzed the $C'$ set of classes described earlier.

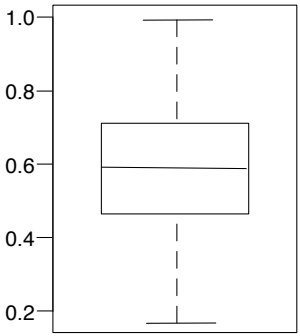


Figure 5.5: Box plot of the invocation inequality grade for the classes in $C'$.

We extend our model with a few new concepts to support this analysis. For every method $m$ we define $f_m$, $a_m$ and $r_m$ to be the location (index) of method $m$ when the list of all methods of the defining class is sorted respectively by frequency of use, alphabetically and randomly[4].

For all methods, we calculate Spearman's rank correlation coefficient (Spearman's coefficient) [MW03] between $f_m$ and $a_m$ and also $f_m$ and $r_m$ for all methods of each class in $C'$. Of course, randomly sorting something by its very nature will give differing results for multiple calculations, so we calculated Spearman's coefficient between $f_m$ and $r_m$ a total of 10 times.

Spearman's coefficient computes agreement between two rankings: two rankings can be opposite (value -1), unrelated (value 0), or perfectly matched (value 1).

The average Spearman's coefficient between $f_m$ and $a_m$ is -0.057414819 (-0.042984891 median) and between $f_m$ and $r_m$ across all classes and all calculations is 0.000521332 (0.006377102 median)

These numbers suggest that, with respect to frequency of use, alphabetical sorting of methods is slightly worse than sorting the methods randomly. Both values are close to zero, meaning that both alphabetical and random sorting are unrelated to frequency of use. Again, we stress that this analysis does not include all the methods of the analyzed classes but just those used throughout the analyzed

[4]Pseudo-randomly as determined by the */dev/random* implementation in Darwin.

projects. Including all the methods would yield results that support our claims even more strongly.

## 5.4 An Improved Way to Organize Documentation

The previous analysis leads to two conclusions:

1. In a majority of API classes, a small number of methods is substantially more frequently used than the rest.

2. Alphabetical ordering is as good as random ordering with respect to the frequency of use of the API of a class

Based on these conclusions, we argue that current documentation browsers can be improved by displaying the subject artifacts sorted according to frequency of use. As a further improvement, the documentation for a class should extend the list of presented methods to include all the commonly invoked methods, even when inherited from superclasses.

To increase the chances that such a change will be adopted by developers, current documentation and code browsing systems should be augmented rather than replaced. Augmenting the way methods are presented rather than replacing the existing alphabetical sorting is preferable as it does not require developers to completely abandon their current knowledge about the documentation.

Such an augmentation for any given set of API classes requires an analysis of the ecosystem to which these classes belong. This analysis needs to be very much like the analysis described in Section 5.2, as it needs to yield exactly the same data for all interesting classes. This data needs to quantify the importance of each method in the context of its class, by counting the number of invocations.

## 5.5 EMF Based Implementation

The client-server architecture of EMF enables different implementations of data presentation. This means that the need for presenting the data in a way that is appealing to the developer can be achieved by implementing a different view. We aim for the data presentation

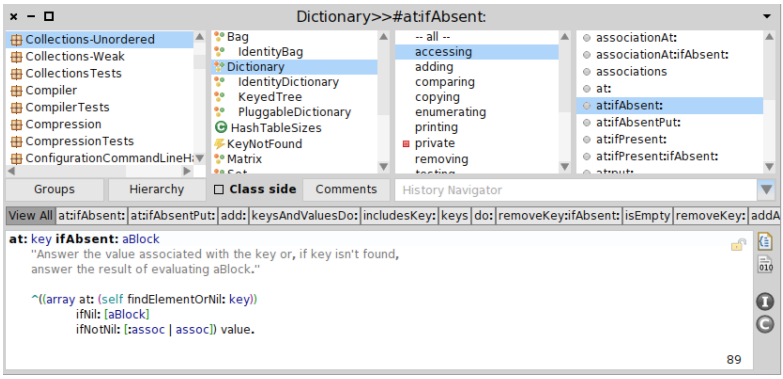


Figure 5.6: A sample presentation of the frequently used methods in Nautilus. The plugin provides the strip of buttons above the method source code. Each button coresponds to a frequently used method.

to be seamlessly integrated with the developers' existing method search process.

In order to provide the developer with the most commonly used methods of the class we first have to determine what those methods are. This is the exact same data that is gathered by the ecosystem-aware type inference back end described in Subsection 4.5.1. Recall that the EATI back end stores for each class-selector combination a selector score *i.e.,* the number of times the selector was sent to an instance of the class. This gives us the option of ranking the selectors of a class by the selector score and getting a list of frequently used methods of that class. This means that we can re-use the EATI back end implementation and just provide a different front end for the same data.

### 5.5.1 The Nautilus Plugin

Smalltalk libraries include the source code so searching for methods is done in the source code itself. The default tool for browsing the source code in Pharo Smalltalk is Nautilus – an advanced implementation of the original Smalltalk system browser [Tes81].

Nautilus provides a framework for developing plugins. As a built in feature of this framework plugins can register to be notified of cer-

tain events by Nautilus. One of these events is triggered when the user selects a class, and information about that class is presented. Our plugin, triggered by this event, consults the data provider module and presents the frequently used methods.

Our initial solution for the way the methods are presented within Nautilus is shown in Figure 5.6. Nautilus is organized in 5 panes, 4 on top half, and one on the bottom. The 4 top panes contain, from left to right, a lists of all packages in the system, a list of classes in a selected package, a list of method protocols[5] in the selected class, and finally a list of methods in the selected protocol. The bottom pane is context sensitive and shows mainly the source code for a selected class or method.

The thin line of buttons between the top and bottom panes is provided by our plugin. Every button corresponds to one frequently used method, and clicking the button opens a new Nautilus window with the desired method selected, and the source code shown. The methods are sorted by frequency from left to right, so that the most popular ones could be read first. The "View all" button opens a separate window with all the frequently used methods shown in a list, similar to the default list of methods in Nautilus.

The general work flow of the plugins is as follows:

1. Detect which class is being viewed by the developer.
2. Consult the data provider module on which are the frequently used methods of that class.
3. Provide that list in a non-intrusive way.

We do not claim there is a correct way to present the data, nor do we claim any approach is superior to others. The question of optimal data presentation is out of the scope of this thesis. The implementations presented in this chapter are only used as a proof of concept.

## 5.6 Observations

To better understand how our approach impacts developers we observed several undergraduate and graduate students using the augmented documentation in their everyday scenarios (course work and

[5]Protocols are convenient groupings of related methods.

research). Several situations in which our approach is directly helpful have been identified in the initial evaluation. To illustrate, we present two cases.

One is the case where a popular method is, due to the alphabetical sorting of methods, located near the end of documentation prolonging the method search process. An example of this is the `select:` method of the `OrderedCollection` class. It is, according to our analysis, the sixth most commonly used method of the class, yet in documentation it is placed on the 54th place out of 62 methods.

The second case is when a popular method of a class is declared higher in the class hierarchy. This leads to the developer wasting time looking through the documentation of a class that does not declare the required method. An example of that is the method for concatenation[6] of `ByteString`s in Pharo Smalltalk. This is, according to our analysis, the most commonly invoked method of the class `ByteString`, yet it is declared in the `SequenceableCollection` class, which is 4 levels higher in the class hierarchy.

## 5.7 Conclusion

In this chapter we present 2 studies performed on data extracted from the usage of API classes used in the QualitasCorpus ecosystem snapshot. The results of these studies indicate two things

1. In a majority of API classes, a small number of methods are substantially more frequently used than the rest.

2. Alphabetical sorting gives unfounded precedence (in the context of searching for methods) to some methods, based on the name of the method rather than its importance or usefulness.

With these two conclusions in mind, we propose an augmentation of the current documentation browsers to also present a small number of the most frequently used methods. We also implemented a proof of concept implementation for the default system browser in Pharo.

We observed developers using the augmented documentation and found use cases in which our approach is beneficial to the developer.

[6]The selector for this method is the comma operator i.e. `'Hello ' , 'World!'`. This is a legal Smalltalk method name.

This shows that the approach has potential, but a larger study of developer usage is needed to confirm and quantify the impact of the approach.

# 6 Ecosystem-Aware Type Guessing

## 6.1 Introduction

Programming languages are usually divided into two groups based on their type system: statically typed languages and dynamically typed languages. Dynamically typed languages are usually considered to be more flexible and more productive [Han10], but lack the explicit type declarations that typical statically typed languages provide. These explicit type annotations are helpful for program comprehension [MHR+12], but can also be used by developer tools (*e.g.,* code completion) to improve the developer experience and productivity.

To partially compensate for the lack of explicit type annotations many Smalltalk developers [GR83] follow a convention of naming method arguments in a way that hints at the expected type of the method argument [Bec97]. This means that it is recommended to name method arguments by prefixing the expected type with the indefinite article "a" or "an". We call this convention "type hints". Listing 6.1 presents the implementation of the *indexOf:* method in the *String* class in Pharo Smalltalk [BDN+09]. The only argument of this method is named *aCharacter* clearly hinting that the method

```
String>>indexOf: aCharacter
    aCharacter isCharacter ifFalse: [^ 0].
    ^ self class
        indexOfAscii: aCharacter asciiValue
        inString: self
        startingAt: 1.
```

Listing 6.1: The implementation of the *indexOf:* method in the *String* class in Pharo Smalltalk. Note that the argument of the method is named *aCharacter* hinting that the expected type is *Character*.

expects to be called with an object of type Character as the parameter.

This convention obviously helps to provide type information about method arguments in a dynamically typed language, but, as any convention not strictly enforced, is only as good as the discipline of developers to follow it.

In this chapter we present a case study of the extent of usage of this convention. Using EMF, we developed a reporting tool that gathered all method arguments from 114 Pharo Smalltalk projects and applied Pharo's built-in system for extracting type information from argument names. We found that this system was able to extract type information from 36.21% of argument names.

Afterwards, guided by the report provided by our tool, we analyzed the argument names that did not yield any type information and, based on the data we observed, developed a few simple heuristics that, when applied, can increase the success rate to 50.69%.

We also note a pattern of expressing so called "Duck-Typed" method arguments *i.e.,* arguments that are expected to be bound to parameters of multiple different types. Almost 1.5% of all method arguments are named in this manner, so any tool attempting to extract type information from method argument names should be aware of this pattern.

To explore whether type hints actually reflect run-time types of arguments we conducted a small study by collecting run-time type information for arguments of two projects and comparing them to their type hints. We find that, on average 76% of type hints reflect run-time types. We discuss the misleading arguments, most of which

would be understood by a developer with domain knowledge, and classify most of them into several patterns.

This chapter is organized as follows: Section 6.2 describes the process of gathering the argument names and Pharo's built-in system for extracting type information from argument names; Section 6.3 discusses the duck-typed method arguments and explains that they are treated as a special case; Section 6.4 discusses the proposed heuristics to improve the success rate; Section 6.5 gives a final overview of the data and conclusions in the previous sections; The study of the correlation between type hints are run-time types is shown in Section 6.6; Section 6.7 discusses future work and finally Section 6.8 concludes.

## 6.2 Data Acquisition

To evaluate the scope of usage of type hints we first gather a large set of method arguments from open source Pharo Smalltalk projects.

We do this by developing a ecosystem-aware tool using EMF. This tool consists of only the back end which extracts to the central database argument names from all methods of all classes in the ecosystem. After that, a post processing step is performed which generates a report as a comma separated file containing a sorted list of the most common argument names in the ecosystem as well as the number of occurrences and list of projects in which the argument name occurs. This report can be used to guide decision making, while the raw data is available in the database for further analyses.

From the 114 projects provided by EMF we extracted a total of 146,297 arguments. We call this set of all arguments *Arg*. To proceed further in our analysis we first need to understand how the process of extracting type information from argument names in Pharo works.

### 6.2.1 The Type Guesser Built into Pharo

The default tool used for extracting type information from argument names in Pharo is part of the code completion tool. The code completion tool is called "NEC" and is based on the eCompletion[1] package developed by Ruben Baker. The process of extracting type information from argument names is referred to as

[1] `http://uncomplex.net/ecompletion/`

"Type Guessing" and is encapsulated in the class side method `getClassFromTypeSuggestingName:` of class `NECVarTypeGuesser`[2].

The implementation of this method is quite simple and is presented in graphical notation in Figure 6.1. This method first removes the leading character of the input argument, and attempts to find and return a class with that name in the system. Failing that, the method removes all characters before the first capital letter of the input arguments, and repeats the attempt. Failing both times, the methods returns `nil`[3].

The edge labels represent the data flow for two example input strings (argument names) *i.e.,* `aCharacter` and `anInteger`.

### 6.2.2 Initial Results

Using the type guesser in Pharo we divided the *Arg* set into two subsets: *Succ* (Successfully guessed) - those from which a type was successfully extracted; and *Fail* (Guess failed) - those that did not yield any type. The definition of these sets is given in Figure 6.2. The function *guessType* is an abstraction of the method `getClassFromTypeSuggestingName`, and returns a set of possible types.

The results of applying `NECVarTypeGuesser` to our data set are shown in Table 6.1. We can see that `NECVarTypeGuesser` guessed the types of fewer than 37% of all arguments. These arguments contain clear type hints and thus are of no further interest to us for further analysis. We continue only on the 63.79% of method arguments that failed to yield a type (the *Fail* set), in an attempt to understand why this is and to improve the success rate.

## 6.3 Duck-Typed Method Arguments

After starting the manual inspection of the method arguments for which `NECVarTypeGuesser` failed to guess a type, we noticed that a substantial number of argument names refer to more than one type. Such method arguments are usually referred to as "Duck-

[2]This class is part of the base Pharo image which can be obtained at `http://get.pharo.org`

[3]`nil` is the Smalltalk equivalent of `null` in Java or `nullptr` in C++

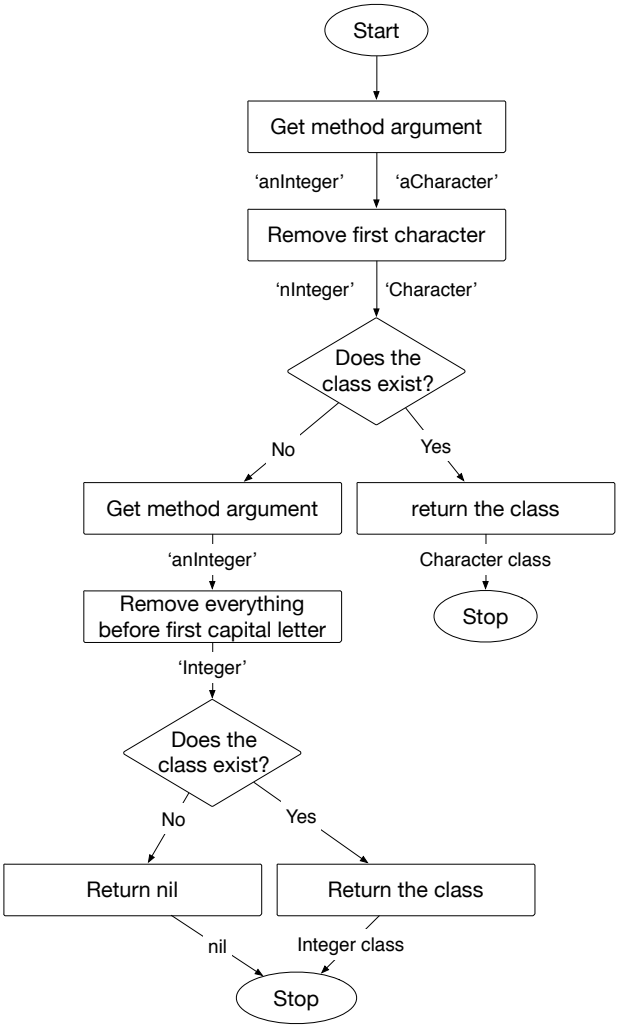


Figure 6.1: An activity diagram of the implementation of `NECVarTypeGuesser»getClassFromTypeSuggestingName`. Edge labels represent data flow for example inputs `aCharacter` and `anInteger`

$$guessType : Arg \rightarrow \{Type\} \tag{6.1}$$

$$Succ = \{a | a \in Arg, guessType(a) \neq \emptyset\} \tag{6.2}$$

$$Fail = \{a | a \in Arg, guessType(a) = \emptyset\} \tag{6.3}$$

Figure 6.2: The core sets. $Arg$ = all arguments, $Succ$ = type guessed, $Fail$ = type not guessed.

| | # | % of $\|Args\|$ |
|---|---|---|
| $\|Arg\|$ | 146,297 | 100% |
| $\|Succ\|$ | 52,981 | 36.21% |
| $\|Fail\|$ | 93,316 | 63.79% |

Table 6.1: The number and percentage of method arguments which do and do not produce a type using `NECVarTypeGuesser`.

Typed"[4] [TFH09]. The term comes from the duck test, attributed to James Whitcomb Riley: *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck"*. In the context of method arguments, this means that the method does not expect a parameter of a particular type, but rather of any type that follows a certain interface. For the sake of simplicity, we consider any method argument that specifies multiple possible types to be duck-typed.

We find that the pattern for expressing that an argument can take on multiple potential types is to concatenate all possible types with the word "Or". For example, one of the most common argument names with this property is `aStringOrByteArray` appearing 99 times in our corpus. To a developer this is a clear message that this argument should be either of type `String` or `ByteArray`, but the implementation of `NECVarTypeGuesser` does not consider this pattern and unsuccessfully attempts to find a class called `StringOrByteArray`.

[4]We acknowledge that all arguments in Smalltalk are potentially "Duck-Typed". We use this term to note the user-specified occurrence of an argument being potentially bound to different types at run time.

### 6.3.1 Impact of Duck-Typed Arguments

To calculate the impact of duck-typed arguments, we extracted all argument names that match the following regular expression.

$$.*Or[A-Z].* \tag{6.4}$$

The word "Or" should be followed by a capital letter to ensure that, due to Camel Notation [WHH11], we only match that word and not words like "Original", "Ordered" *etc.* A total of 2139 method arguments matched this regular expression.

Manual inspection revealed that 36 occurrences of the extracted arguments do not in fact refer to multiple types. These are all occurrences of two different argument names: `aBlockWithZeroOrOneParameter` and `aZeroOrOneArgBlock`. It is obvious that these argument names refer to a varying number of arguments of a `BlockClosure` *i.e.,* Lambda expression, and not multiple possible types.

So the very simple regular expression based heuristic we used thus far can easily be integrated in any tool, and on our data set has a false positive rate of only 1.68%. We consider all arguments whose name match the regular expression from Equation 6.4 but are not duck-typed to be false positives *i.e.,* 36 occurrences of `aBlockWithZeroOrOneParameter` and `aZeroOrOneArgBlock`. A more complex heuristic based on natural language processing could also be an option.

With this in mind, we define the set of duck-typed method arguments (*Duck*) in Figure 6.3 as the set that matches our regular expression from Equation 6.4 excluding `aBlockWithZeroOrOneParameter` and `aZeroOrOneArgBlock`. The *name* function extracts the name of the argument as a string.

The *Duck* set contains a total of 2,103 method arguments or 1.44% of the *Arg* set. This is not an insignificant percentage and any tool attempting to guess types should be aware of the existence of this pattern.

### 6.3.2 Distribution of Number of Types in Duck-Typed Arguments

A followup question regarding duck-typed method arguments is how many different types are hinted at in these arguments.

$$
\begin{aligned}
Duck = \{a | a \in Fail,& \\
& regexMatch(a, \text{“}. * Or. * [A - Z]\text{”}), \\
& name(a) \neq \text{“}aBlockWithZeroOrOneParameter\text{”}, \\
& name(a) \neq \text{“}aZeroOrOneArgBlock\text{”}\}
\end{aligned}
\tag{6.5}
$$

Figure 6.3: The *Duck* set contains all arguments that hint at multiple types.

The distribution is presented in Table 6.2. As per intuition, the vast majority (94.82%) of duck-typed arguments hint at two types (*e.g.,* `aStringOrByteArray`, `aUrlOrString`, `aStringOrText`). Around 7% hint at three types (*e.g.,* `aStringOrCollectionOrBlock`, `aDateOrNumberOrString`), and less than half of a percent hint at more. The maximum number of different hinted types is 5, with the argument name `aSelectorOrElementOrjQueryOrBooleanOrNumber` [*sic*]. This might be considered an unwieldy argument name, but on the other hand, it completely defines the number of expected types within the source code with no need for additional documentation.

| Different hinted types | Number of occurrences | % (of $\|Arg\|$) | % (of $\|Duck\|$) |
|---:|---:|---:|---:|
| 5 | 5 | 0.00% | 0.24% |
| 4 | 5 | 0.00% | 0.24% |
| 3 | 149 | 0.10% | 7.09% |
| 2 | 1994 | 1.36% | 94.82% |

Table 6.2: Distribution of the number of different types in duck-typed method arguments.

## 6.4 Heuristics for Type Hints

We continue the manual inspection of arguments that failed to yield a type on the set *Fnd* defined as

$$Fnd = Fail \setminus Duck \tag{6.6}$$

This is a set of all arguments from the *Fail* set that are not duck-typed (*Fnd* — failed, non duck-typed). It contains 91,213 arguments which is 62.35% of the *Arg* set or 97.75% of the *Fail* set. The aim of further inspection is to identify subsets of *Fnd* that contain type hints, and identify heuristics for identifying the types.

### 6.4.1 *spec* and *html*

In the *Fnd* set we find a frequent occurrence of the arguments `spec` and `html`.

Inspection of the source code reveals that `spec` is the standard name used for specifications of Metacello versions. Metacello is a package management system for Monticello, a distributed version control system for Smalltalk. The implementation details are not important, but we can claim that the arguments named spec are of type `MetacelloAbstractVersionConstructor`, as it is the superclass for all classes used to construct Metacello versions. So we define the *Spec* set as

$$Spec = \{a | a \in Fnd, name(a) = \text{“}spec\text{”}\} \tag{6.7}$$

This set contains 6,132 elements or 4.19% of the *Arg* set.

A similar situation exists with the arguments named `html`. A common practice when writing applications using Seaside [DLR07], a web development framework for Smalltalk, is to pass the object representation of the HTML element as an argument to methods of objects that perform an action on it (usually the object renders itself on the HTML element). All of these arguments are instances of `WAHtmlCanvas`. As with the `spec` argument name we define the *Html* set (which contains 2,935 elements or 2.01% of the *Arg* set) as

$$Html = \{a | a \in Fnd, name(a) = \text{“}html\text{”}\} \tag{6.8}$$

### 6.4.2 Blocks, Strings and Collections

Using block closures in Smalltalk is common practice. The class `BlockClosure` offers the default implementation. Unfortunately for the default type guessing algorithm, a majority of method arguments that are expected to be an instance of `BlockClosure` are

not named `aBlockClosure` (only 47 occurrences of arguments named `aBlockClosure` in *Arg*) but rather `aBlock` (6,167 occurrences of arguments named `aBlock` in *Arg*) as Figure 6.4 summarizes. Using `aBlock` rather than `aBlockClosure` is even present in the book "Smalltalk Best Practice Patterns" [Bec97] by Kent Beck.

To give more context to arguments hinting at `BlockClosure`, developers often add additional descriptors to the argument name. Examples of such argument names are `toBlock`, `fromBlock`, `anErrorBlock`, `aOneArgBlock`, `aFormatBlock` *etc.* Also, a substantial number of arguments are named simply `block`, ignoring the article.

In order to group all these different ways of specifying arguments of type `BlockClosure` we define a set called $Block'$ as the set of all arguments whose name matches the regular expression ".$*(B|b)lock.*$". This is formally defined in Figure 6.5 Equation 6.12.

Following the same logic we define two more sets. The first attempts to group all arguments hinting at the `Collection` type — *Coll* (Figure 6.4 Equation 6.13) and the second for arguments hinting at the `String` type — *String* (Figure 6.4 Equation 6.14).

The number of elements in all of these sets can be found in Table 6.3. It is worth noting that the set $Block'$ contains over a thousand more elements than set *Block*, showing that using the simple heuristic can attach a type to a much larger set of arguments.

| | # | % ($\|Arg\|$) |
|---|---|---|
| $\|Block'\|$ | 7,886 | 5.39% |
| $\|Coll\|$ | 559 | 0.38% |
| $\|String\|$ | 1,793 | 1.23% |

Table 6.3: The cardinalities of the $Block'$, *Coll* and *String* sets.

### 6.4.3 Duplicate Entries in Sets *Block′*, *Coll* and *String*

The regular expression based definitions of sets $Block'$, *Coll* and *String* are quite naive, and further inspection of the elements of these sets reveals that certain arguments appear in multiple sets as

$$|\{a|a \in Arg, name(a) = \text{“}aBlockClosure\text{”}\}| = 47 \tag{6.9}$$

$$Block = \{a|a \in Arg, name(a) = \text{“}aBlock\text{”}\} \tag{6.10}$$

$$|Block| = 6,167 \tag{6.11}$$

Figure 6.4: Many more arguments are named `aBlock` than `aBlockClosure`

shown in Figure 6.5 Equation 6.17. The problem arises in argument names that match multiple regular expressions.

The number of such elements is fairly small. A total of 18 arguments appear in the *String* and *Coll* sets, and all are named `aCollectionOfStrings`. Their name is a clearly hinting at the type `Collection` rather than `String`. Similarly, 12 arguments appear in the *Block′* and *String* sets, and are all named `aBlockAnsweringAString`, hinting at the type `BlockClosure` and not `String`. No arguments are present in both *Block′* and *Coll* sets.

With this in mind we can conclude that, in the *Fnd* set, a clear rule can be observed for dealing with these ambiguities. We notice that all arguments that appear in these three sets can be placed in the adequate set by following a strict hierarchy of set priorities: blocks are higher priority than collections which are higher priority than strings.

Namely, all duplicate arguments from the *Coll* set, are properly placed, since the duplicates named `aCollectionOfStrings` are clearly collections, thus collections have a higher priority than strings. Similarly, all duplicate arguments from the *Block′* set, are properly placed, since the duplicates named `aBlockAnsweringAString` are clearly blocks. Thus blocks have a higher priority than strings. We artificially introduce the rule that blocks are higher priority than collections in order to make our heuristic complete. This rule might cause false positives in the cases such as the hypothetical argument name `aCollectionOfBlocks`, placing such an argument in the set of blocks rather than in the set of collections where it would belong.

Another approach to removing duplicate entries requires a more thorough analysis of these cases, either by natural language processing techniques or by focusing on splitting the argument name

$$Block' = \{a|a \in Fnd, \\ regexMatch(a, ".*(b|B)lock.*")\} \tag{6.12}$$

$$Coll = \{a|a \in Fnd, \\ regexMatch(a, ".*(c|C)ollection.*")\} \tag{6.13}$$

$$String = \{a|a \in Fnd, \\ regexMatch(a, ".*(s|S)tring.*")\} \tag{6.14}$$

$$Coll' = \{a|a \in Fnd/Block' \\ regexMatch(a, ".*(c|C)ollection.*")\} \tag{6.15}$$

$$String' = \{a|a \in Fnd/(Block' \cup Coll'), \\ regexMatch(a, ".*(s|S)tring.*")\} \tag{6.16}$$

$$\begin{aligned} Block' \cap Coll &= \emptyset \\ |Block' \cap String| &= 12 \\ |String \cap Coll| &= 18 \end{aligned} \tag{6.17}$$

$$\begin{aligned} Block' \cap Coll' &= \emptyset \\ Block' \cap String' &= \emptyset \\ String' \cap Coll &= \emptyset \end{aligned} \tag{6.18}$$

Figure 6.5: Sets of arguments based on regular expressions

$$DuckF = Duck \backslash DuckS \tag{6.19}$$

Figure 6.6: The *DuckF* set contains all duck-typed arguments whose type could not be guessed.

by "Of" and determining the priorities by the order. We feel this would introduce a lot of complexity for not much gain and leave out such attempts.

We apply these hierarchy rules in defining the sets $Coll'$ (Figure 6.5 Equation 6.15) and $String'$ (Figure 6.5 Equation 6.16), and ensure that there is no overlap between these sets (Figure 6.5 Equation 6.18).

### 6.4.4 Guessing Types of Duck-Typed Arguments

All the sets defined in this section thus far are subsets of *Fnd*, meaning that the arguments we defined as duck-typed are not included in any of the sets. To determine the set of duck-typed arguments whose type can be guessed ($DuckS \subset Duck$) we follow a simple algorithm:

1. Split the name of the argument $a \in Duck$ by the keyword *Or*
2. Treat each of the sub-names as a valid name of a hypothetical argument $b$
3. If $guessType(b) \neq \emptyset$ then we guessed the type of $a$, and $a \in DuckS$.
4. If not, assume that $b \in Fnd$
5. If it holds that
   $b \in Spec \vee b \in Html \vee b \in Block' \vee b \in Coll' \vee b \in String'$ then we guessed the type of $a$, and $a \in DuckS$.
6. If not, $a \notin DuckS$

Essentially, if we can guess one of the types that the argument name hints at, we declare the argument type guessed. So the set *DuckS* holds all arguments from the set *Duck* whose type can be guessed. The remaining arguments make up the *DuckF* set defined in Figure 6.6. The cardinality of these sets is given in Table 6.4.

| | # | % ($\|Arg\|$) |
|---|---|---|
| $\|DuckS\|$ | 1,905 | 1.30% |
| $\|DuckF\|$ | 198 | 0.14% |

Table 6.4: The cardinality of the *DuckS* and *DuckF* sets.

## 6.5 Final Results

With all the heuristic based sets defined in Section 6.4 we have exhausted the ways in which we can guess types in the *Arg* set. The potential for other heuristics still exists, *e.g.,* arguments named `index` can be considered to be integers, plural nouns can be considered collections *etc.* but without additional studies dedicated to these situations we cannot claim that these potential heuristics are well-reasoned.

The set $H$, defined in Figure 6.7 Equation 6.20, is the union of all sets defined by heuristics and accounts for 13.18% of all arguments. Now, we can finally define a set of all arguments whose type can be guessed from the name. We call this set $Succ'$ and define it in Figure 6.7 Equation 6.21. Also, in Figure 6.7 Equation 6.23, we define the set $Fail'$, as the set of all arguments whose types can not be guessed from the name. It is defined as the union of all duck-typed arguments whose type is not guessable ($DuckF$) and all non-duck-typed arguments whose type is not guessable ($F$).

The cardinality of these sets, as well as all their subsets is presented in Table 6.6. We can see that the default type guessing implementation can be substantially improved by incorporating the proposed heuristics. The total number of arguments whose type can be guessed is 74,161 or 50.69% of all arguments. This set is by no means complete. If we look at just the top ten most frequent names of arguments from the $F$ set show in Table 6.5, we can see that there are still argument names that contain hints *i.e.,* `aName`, `aBrick` and `aValue`. These hints are more delicate and might carry a lot more meaning for a developer with domain knowledge, but providing tool support for such cases is a more challenging task. With all this in mind, we can thus conclude that type hints are a commonly used way to name method arguments in Smalltalk projects, and that even

| Argument name | # | % ($\|Arg\|$) |
|---|---|---|
| `n` | 1511 | 1.03% |
| `aName` | 1440 | 0.98% |
| `a` | 1118 | 0.76% |
| `lda` | 1092 | 0.75% |
| `nodes` | 868 | 0.59% |
| `aBrick` | 825 | 0.56% |
| `work` | 816 | 0.56% |
| `aValue` | 786 | 0.54% |
| `info` | 753 | 0.51% |
| `evt` | 670 | 0.46% |

Table 6.5: The top ten most frequent argument names in the $F$ set.

$$H = Spec \cup Html \cup Block' \cup Coll' \cup String' \tag{6.20}$$

$$Succ' = Succ \cup H \cup DuckS \tag{6.21}$$

$$F = Fnd \backslash H \tag{6.22}$$

$$Fail' = F \cup DuckF \tag{6.23}$$

Figure 6.7: The $H$ set contains all arguments all sets defined by heuristics in Section 6.4, the $Succ$'set contains all arguments whose type is guessable and $Fail'$ contains all arguments whose type is not guessable.

fairly simple tool support can work about 50% of the time.

## 6.6 Quality of Type Hints

So far we have focused on the quantity of type hints in a large number of Smalltalk projects. In this section we conduct a separate analysis on the quality of type hints in two Smalltalk projects: Glamour [BGR+09], a framework for describing navigation flow of GUI data browsers; and Roassal [ABC+13], a visualization engine

| | | | | # | % $\lvert Args \rvert$ |
|---|---|---|---|---|---|
| $\lvert Arg \rvert$ | | | | 146,297 | 100% |
| $\lvert\rightarrow$ | $\lvert Succ' \rvert$ | | | 74,161 | 50.69% |
| $\lvert$ | $\lvert\rightarrow$ | $\lvert Succ \rvert$ | | 52,981 | 36.21% |
| $\lvert$ | $\lvert\rightarrow$ | $\lvert DuckS \rvert$ | | 1,905 | 1.30% |
| $\lvert$ | $\lvert\rightarrow$ | $\lvert H \rvert$ | | 19,275 | 13.18% |
| $\lvert$ | | $\lvert\rightarrow$ | $\lvert Block' \rvert$ | 7,886 | 5.39% |
| $\lvert$ | | $\lvert\rightarrow$ | $\lvert String' \rvert$ | 1,793 | 1.92% |
| $\lvert$ | | $\lvert\rightarrow$ | $\lvert Coll' \rvert$ | 559 | 0.38% |
| $\lvert$ | | $\lvert\rightarrow$ | $\lvert Spec \rvert$ | 6,132 | 4.19% |
| $\lvert$ | | $\lvert\rightarrow$ | $\lvert Html \rvert$ | 2,935 | 2.01% |
| $\lvert\rightarrow$ | $\lvert Fail' \rvert$ | | | 71938 | 49.31% |
| | $\lvert\rightarrow$ | $\lvert DuckF \rvert$ | | 198 | 0.14% |
| | $\lvert\rightarrow$ | $\lvert F \rvert$ | | 71,938 | 49.17% |

Table 6.6: The cardinality of all defined sets. Hierarchy represents subset relation.

for Smalltalk. The reason we chose these two projects for our analysis is their rich example library which enables us to easily run a dynamic analysis similar to real-world usage of these projects. The end goal of this analysis is to verify whether types extracted from type hints match run-time types of arguments.

### 6.6.1 Acquisition of Run-Time Types

In order to collect run-time types of method arguments in our case projects we instrumented the source code of these projects using a slightly modified version of a tool called "Variable Tracker"[5] used for gathering run-time type information from Smalltalk projects. The modification was to limit the tool to method arguments only.

After instrumenting the source code, we executed the examples for each project. Glamour defines 68 examples of which one failed to execute. Roassal defines 63 examples of which four failed to execute. This is by no means an exhaustive dynamic analysis, but

[5] `http://smalltalkhub.com/#!/~rostebler/VariableTracker`

it does provide a usage similar to real world applications of these frameworks.

The result of this is presented in the "Total arguments" entry of Table 6.7. As the table shows, we collected run-time type information on 251 and 559 arguments from Glamour and Roassal respectively.

During normal execution some of these arguments might get multiple different types due to Smalltalk's dynamic type system. We did not encounter such situations during our dynamic analysis. A more thorough dynamic analysis might yield such cases.

| | Glamour | Roassal |
|---|---|---|
| Total arguments | 251 | 559 |
| Contain type hint | 141 | 159 |
| Good type hints | 126 | 103 |
| Misleading type hints | 15 | 56 |

Table 6.7: A summary of the dynamic analysis results.

### 6.6.2 Type hints and run-time types

To assess the quality of type hints we first extract the arguments that contain type hints. We include all arguments that contain type hints according to the type guesser described in Subsection 6.2.1 as well as all those that match the heuristics described in Section 6.4. The result is presented in the "Contain type hint" entry in Table 6.7. A total of 141 and 160 arguments contain type hints in Glamour and Roassal.

Finally, we separate the arguments whose names contain type hints that match the run-time type or one of its subclasses. We include the subclasses of the run-time type in order to include occurrences of subtype polymorphism [CW85]. A typical toy example would be an argument named `anAbstractShape` getting the type `Circle` (a subclass of `AbstractShape`) at run time. The results of this separation is shown in the "Good type hints" entries in Table 6.7. We can see that in Glamour 89.36% of type hints are good, meaning the guessed type or one of its subtypes is used at run time. In Roassal this number is 64.77% and the average across both applications

is 76%.

### 6.6.3 Misleading Type Hints

The last entry in Table 6.7 ("Misleading type hints") shows the number of arguments whose type hint did not match the run-time type. There is a total of 71 such arguments, 15 from Glamour and 57 from Roassal. Many of these arguments are misleading to our tools, but might not be to a developer with more context and domain knowledge. An overview of all these arguments is presented in Table 6.8.

We can notice a few distinct patterns in this data, and their description follows. All but 11 arguments (15.49%) fall into one of these patterns.

#### Class Conflict

This situation emerges when the guessed type exists, but the run-time type has a similar name and no hierarchical connection. Examples of this are

- Arguments that hint at the type `Browser` (a class present in the default Pharo image) but at run time receive `GLMBrowser` (GLM stands for Glamour)
- Arguments that hint at the type `Shape` or `Canvas` (both classes present in the default Pharo image) but at run time receive some subclass of `TRShape` or `TRCanvas` (TR stands for Trachel, a module used in Roassal) *etc.*

This pattern accounts for 36.62% (26 out of 71) arguments with misleading type hints.

#### Any Object Data Model

Roassal is a very flexible visualization engine and can visualize any set of objects and their connections. To make this possible, it relies on using any object as a data model for generating the visualization, and specifying through dynamic features of Smalltalk *i.e.,* metaprogramming, how to interact with the model. That is why it is common to find arguments named `aModel` that get bound to many different types at run time. The problem comes due to the fact that `Model` is a class in the default Pharo image. In our set of misleading type hints this pattern accounts for 8.45% (6 out of 71) arguments.

| Project | Argument name | Run-time class | Pattern | # |
|---|---|---|---|---|
| Glamour | `aBrowser` | `GLMBrowser` | Class conflict | 12 |
| | `anAnnouncement` | `Announcement class` | - | 2 |
| | `aSymbolOrABlock` | `ByteString` | - | 1 |
| Roassal | `aModel` | `ByteString`, `SmallInteger`, *etc.* | Any object data m. | 6 |
| | `toBlock`, `fromBlock`, `followBlock` | `ByteSymbol` | Block or symbol | 13 |
| | `aValueOrOneArgBlock` | `Color`, `SmallInteger`, *etc.* | Block or value | 11 |
| | `aValueOrASymbolOrOneArgBlock` | `SmallInteger` | Block or value | 2 |
| | `trachelShape`, `anotherShape`, *etc.* | `subclass of TRShape` | Class conflict | 11 |
| | `trachelCanvas` | `TRCanvas` | Class conflict | 2 |
| | `aBehaviour` | `TRNoBehaviour` | Class conflict | 1 |
| | `aFloat` | `SmallInteger` | Convertable | 2 |
| | `aKey` | `ByteSymbol` | - | 2 |
| | `aBlock` | `SmallInteger` | - | 1 |
| | `aLineShapeOrBlock` | `TRLine class` | - | 1 |
| | `aTRLine` | `TRBezierLine` | - | 1 |
| | `aTRShape` | `TRLine class` | - | 1 |
| | `depClass` | `String class` | - | 1 |
| | `listOfElementsOrOneArgBlock` | `TRGroup` | - | 1 |

Table 6.8: Overview of all argument names that have misleading type hints.

### Block or Value

In 18.3% (13 out of 71) cases of misleading types we notice a duck-typed argument that starts with `aValue` and also hints at expecting a `BlockClosure`. A "Value" in this case can be any object, and in case a `BlockClosure` is provided, it is assumed that evaluating it will produce the expected value. A more exhaustive dynamic analysis might produce cases where this argument is bound to an instance of `BlockClosure`. This pattern is very similar to "any object as a data model" but we separate them because `Model` is a class in the Pharo image, and `Value` is not. The type hint in this pattern comes from the hinted option of using instances of `BlockClosure`.

The reason why Roassal can expect any object as a value is that it extends the class `Object`[6] with the method `rtValue:` which is the only method invoked on arguments from this pattern. The default implementation of this method just returns the receiver object, but it is overridden in the `BlockClosure` class to evaluate the closure with the method arguments.

### Block or Symbol

A common idiom in Smalltalk source code is that methods that expect an instance of `BlockClosure` can also accept an instance of `Symbol` [BDN+09].

This is illustrated well by the filtering method `select:` of the collections package in Smalltalk. This method expects an instance of `BlockClosure` that describes the criterion for selection of elements of the collection. If this criterion is to only invoke a single method with no arguments (for example, the `isZero` method of class `Number`), than we can provide just the selector (method name) as an instance of `Symbol`. The result is smaller code that is more readable. This idiom exploits duck-typing by implementing the `value:` method — normally associated with block closures — for the `Symbol` class in the obvious way.

This pattern accounts for 18.3% (13 out of 71) of the misleading type hints. As with the "Value or Block" pattern, a more thorough dynamic analysis would most likely find cases where these arguments would be bound to instances of `BlockClosure`.

[6] All classes in Smalltalk eventually inherit from the `Object` class

**Convertible**

We found only two occurrences of this pattern, but it is a situation that can theoretically happen much more often. Essentially, the argument was hinting at expecting an instance of the class `Float`, but received an instance of the class `SmallInteger` at run time. The `SmallInteger` class is not a specialization of the `Float` class, but implicitly, because $\mathbb{N} \subset \mathbb{R}$, any integer is also a float.

The usage of this argument is restricted to arithmetic operations making the instance of `SmallInteger` completely indistinguishable from an instance of `Float`. This is because all the arithmetic operations in Pharo provide an implicit coercion to the appropriate type when necessary *i.e.,* before invoking the VM primitive that performs the calculation.

## 6.7 Future Work

We envision three directions of future work: continuously monitoring the way arguments in given projects change over time (Subsection 6.7.1); attempting to integrate dynamic analysis to infer patterns between argument names and their run-time types (Subsection 6.7.2); and performing a comparison to type inference engines (Subsection 6.7.3).

### 6.7.1 Continuous monitoring

Code evolves, and with time new patterns in type hints might appear. A heuristic based type guessing tool would be only as good as its heuristics, so continuously monitoring argument names from projects of interest could identify emerging trends in argument names, and notify the tool developer of the new patterns. This would prompt the developer to investigate the new patterns, develop an appropriate heuristic and integrate it into the tool. Since the tool used to guide the study presented in this chapter is based on EMF, the data is automatically periodically refreshed, and a new report is generated. This means that supporting this future work is implicit in EMF.

### 6.7.2 Dynamic analysis

An alternative to observing static changes in source code could be observing a system in question at run time, and attempting to find patterns of mappings between argument names and run-time types. This could be done automatically through machine learning techniques, and could be tailored to the project domain helping with some of the bad type hints from Section 6.6 *i.e.,* an argument named `aShape` can hint at the `Shape` class or `TRShape` class, and run-time information can distinguish which.

### 6.7.3 Comparison with Type Inference Engines

The standard approach for statically obtaining type information in dynamically typed languages is to perform type inference. Many different type inference approaches exist [Mil78, CF91] and some approaches have existing implementations in Pharo [SLN14a, PMW09, MBGN16]. The question is whether type hints or type inference can provide more type information for method arguments in Smalltalk, or whether their combination is worth the effort.

## 6.8 Conclusion

In this chapter we present two case studies, one on how frequently type hints are used in method argument names in Smalltalk and one on the quality of those type hints in relation to run-time types.

In the first study we analyze a total of 146,297 arguments taken from 114 Pharo Smalltalk projects, and conclude that the existing tool for type guessing has a 36.21% success rate. We manually analyze the arguments that failed to yield a type, and propose several heuristics that improve the percentage of guessed types to 50.69%. This percentage is not final, as many other heuristics potentially exist, but further inquiry and more domain knowledge is required to formulate them. So the main conclusion of the study is that at least one in two method arguments in Smalltalk projects contains a useful type hint. Another conclusion drawn from this case study is that 1.44% of method arguments hint at multiple types. We present a very simple heuristic for identifying such method arguments with a false positive rate of only 1.68% in our data set.

In the second study we collected run-time types of method arguments from two projects with a rich set of examples used as input, and compared the run-time types to the types extracted from type hints. We find that in 76% of cases the type hint matches the run-time type when controlling for subtype polymorphism. We also present an analysis of the misleading type hints, and identify several patterns that better our understanding of why the type hints are misleading.

We propose several directions of future work, most on improving Smalltalk tools for type guessing using the information from the conducted studies. Beside that, replicating these studies with different languages would help broaden and generalize the understanding of type hints.

# 7 The Object Repository

## 7.1 Introduction

Every object-oriented language includes a mechanism for creating new objects, and developers leverage this mechanism to create complex objects needed in software systems. Most classes contain a default constructor, a way to create a default instance of that class, but some classes do not have a default representation, and are instantiated through more complex construction mechanisms such as parametrized constructors, factory methods, the builder design pattern and so on. Fully formed objects can also be created through any valid object usage protocol, which may include an arbitrary number of interacting objects and method invocations.

Finding a way to properly instantiate classes can be non-trivial, yet, code snippets that instantiate these classes exist throughout their client classes. Existing approaches for mining code snippets provide useful information about object creation and usage, but they lack the feature of being executable which is important to enable approaches that require objects on demand.

We propose to mine available software projects for code snippets that instantiate classes, henceforth referred to simply as snippets.

We aim to create a repository of such snippets, which can be used to facilitate several software engineering tasks such as augmenting documentation, new testing approaches, support for program comprehension and others.

We realize this approach by extracting all AST nodes from all methods of all available classes, converting them to their source code representation and attempting to execute them. If the execution is successful, *i.e.,* produces an object, we save the snippet in a database and associate it to the type of the produced object.

We implemented this approach using EMF and applied it to 141 open source Pharo projects and a selection of classes contained in the base Pharo image[1]. We find that the result of this approach is that around 10% of AST nodes, when converted to source code and executed, produce objects for almost 80% of all the analysed classes. We check several aspects of the snippets to better understand their properties and also analyse the nodes that failed to produce objects, and discuss how to tackle the reasons for the failures.

The chapter is organised as follows: Section 7.2 discusses potential uses of the Object Repository, Section 7.3 describes the approach, the formal set model and discusses implementation details, Section 7.4 describes the results of applying the formal model to the Pharo projects and the analysis of the produced data, Section 7.5 discusses future work, and finally Section 7.6 concludes.

## 7.2 Motivation

This section presents some of potential use cases for the Object Repository. Though there are several software engineering tasks that can benefit from the availability of such a repository, our discussion advocates three that we found most applicable.

### 7.2.1 Software Documentation

Documentation, when available and up to date, is still the most reliable and widely used resource for understanding software systems and APIs. Much work has been done around the idea of mining usage examples to enrich documentation [BW12, ZXZ+09, HM05],

[1]The classes in the base image are similar to the standard library in languages such as C++ or Java.

however, none of the example snippets given are usable to directly create objects. Code snippets from the Object Repository could be used as a complementary source for such examples, with the additional knowledge that the snippets are immediately executable. The main requirement for this use case is to have concise and representative snippets.

Also, since the snippets in the Object Repository produce objects, one could introduce a concept of a "playground" within the documentation where a developer could experiment with a given live instance of the class whose documentation she is reading. Many similar "playgrounds" exist for languages such as Go[2] and Haskell[3], allowing developers to simply try out parts of the language.

This requires at least one snippet associated with the documented class. Unlike the previous case, the quality of snippets is of no importance, as the user is only meant to interact with the object rather then the snippet which created it.

### 7.2.2 Software Testing

Modern approaches to inspecting objects rely on object specific representations [CGNS15]. This means that the author of a class, or anyone else through extensions, can specify a way that the object can visually represent itself to the user of the object inspector. The object inspectors provide the user all available ways to represent the object, and the user chooses one that suits the current context.

According to our discussion with researchers in the field, testing new representations is laborious since they are usually required to create the objects manually. Alternatively, the Object Repository could provide a set of objects gathered from the ecosystem as a test suite that enables testing if a new representation of an object is stable automatically.

Testing software by generating random test inputs is a well researched field [DN81, PLEB07, CKMN03]. Integrating objects from the Object Repository with the random input generated by these approaches could help cover the corner cases that are hard to detect with raw random testing by focusing the attention on the part of the search space that is more representative of real world usage.

[2] `https://play.golang.org/`
[3] `https://tryhaskell.org/`

Moreover, starting from actual instances from the Object Repository instead of, or in combination with, random ones could improve the results. For instance in case of genetic algorithms [MMS01], this can guide the genetic algorithm to an acceptable population of input data much faster while also avoiding local maximums.

Method arguments are usually checked for validity at the beginning of a method. In case the argument is not valid (*e.g.,* defies the contract or the preconditions), the method should signal this fact to the caller in a expected manner *e.g.,* by throwing an exception, or returning an error value. In order to test this a developer would require multiple instances of the argument type both valid and invalid in the context of being input for that method. Assuming that arguments of the method are of a type that is present throughout the ecosystem, the Object Repository should contain code snippets needed to create such instances. This facilitates, to some extent, verifying automatically that the validation of the method arguments behaves as expected.

To realize these use cases the Object Repository should contain as many snippets associated with a class as possible. The snippets should also produce representative and diverse objects.

### 7.2.3 Software Evolution and Maintenance

While studying source code is the main way that developers interact with programs [KDV07, KBR14] many program comprehension tasks require runtime observation [LM10a, LM10b]. Nevertheless, running a system and placing it in a desired state can be challenging for several reasons like lack of input to the system, lack of knowledge about the system, long system running time before reaching a desired point.

Having a way to create a live object of a required type could spawn a running system at any point in the source code by filling all the gaps in the execution context with blank objects of the adequate type. These objects should be presented in an object-inspector-like interface allowing the developer to set the values of these context objects and guide the execution of the program, as one would do in a debugging session. One example of one such usage could be in the domain of application security. Executing parts of source code in a sandbox created from objects taken from the Object Repository could ensure that the execution does not have any unexpected side

effects.

Objects taken from the Object Repository could be used to help disambiguate results of type inference engines for dynamically typed languages [SLN14a]. Many of these type inference engines provide a list of potential candidate types for a variable. Thanks to the Object Repository, having instances of those types, and attempting to execute the code with each of those objects assigned to the variable in question could shed some light on the types which are more likely false positives.

The main requirement for these use cases is to have an Object Repository that contains snippets associated to as many classes as possible, extending the applicability of this use case to more project.

## 7.3 The Approach

We aim to mine code snippets from projects by transforming all available methods into their abstract syntax tree (AST) representation, transforming each AST node[4] into its source code representation, attempting to execute it and observing the return value of the execution. We ignore this code snippet if the execution fails to compile, or to produce a return value. The ones that return an object are saved to the Object Repository and associated to the type of object produced by their execution.

For example, Listing 7.1 shows a method from the *Pomodoro* project[5]. This method checks if an instance variable `progressBar` is `nil` (line 2) and, if so, assigns it a new instance of `ProgressBarMorph` (line 3). Finally, it returns this instance variable (line 5).

```
1 PomodoroMorph>>progressBar
2   progressBar ifNil: [
3     progressBar := ProgressBarMorph new
4   ].
5   ^ progressBar
```

Listing 7.1: Example method used to illustrate the approach.

[4]Each AST node is technically an AST tree *i.e.,* the subtree of the AST of the method with the node in question at the root. We use the term "AST node" in place of "AST subtree with the node at the root" for simplicity.

[5]`http://smalltalkhub.com/#!/~TorstenBergmann/Pomodoro`

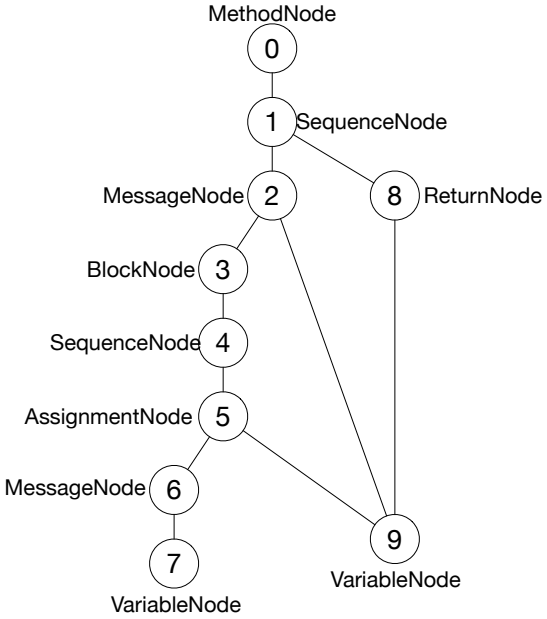


Figure 7.1: Abstract syntax tree of the method in Listing 7.1.

We first parse this method and build the AST. Figure 7.1 shows a graphical representation of this AST with all the nodes indexed and the type of the node shown next to it. We then transform each node into a source code snippet, and attempt to execute it.

Table 7.1 summarizes the results. This table represents each AST node by index, the type of the node, the source code representation of the node, and finally the type of the value obtained by executing the snippet. We can see from this table that out of the 9 AST nodes only 2 are, when transformed into source code, executable and produce objects.

### 7.3.1 Formal Model

To better explain the proposed approach, we introduce a small set-theoretical model. This subsection discusses domains and sets used in this model, the function that models the execution of the snippets,

| # | Node Type | Corresponding Snippet | Result Type |
|---|---|---|---|
| 1 | SequenceNode | `progressBar ifNil:[progressBar:=ProgressBarMorph new].`<br>`^ progressBar` | - |
| 2 | MessageNode | `progressBar ifNil:[progressBar:=ProgressBarMorph new]` | - |
| 3 | BlockNode | `[progressBar:=ProgressBarMorph new]` | - |
| 4 | SequenceNode | `progressBar:=ProgressBarMorph new` | - |
| 5 | AssignmentNode | `progressBar:=ProgressBarMorph new` | - |
| 6 | MessageNode | `ProgressBarMorph new` | `ProgressBarMorph` |
| 7 | VariableNode | `ProgressBarMorph` | `Metaclass` |
| 8 | ReturnNode | `^ progressBar` | - |
| 9 | VariableNode | `progressBar` | - |

Table 7.1: The results of applying our proposed approach to the example method from Listing 7.1.

$$C : \textit{Domain of classes} \tag{7.1}$$

$$M : \textit{Domain of methods} \tag{7.2}$$

$$def_m : M \to C \tag{7.3}$$

$$N : \textit{Domain of AST nodes} \tag{7.4}$$

$$def_n : N \to M \tag{7.5}$$

$$S : \textit{Domain of snippets} \tag{7.6}$$

$$toCode : N \to S \tag{7.7}$$

$$O : \textit{Domain of objects} \tag{7.8}$$

$$O_0 = O \cup \emptyset \tag{7.9}$$

$$instanceof : O \to C \tag{7.10}$$

$$execute : S \to O_0 \tag{7.11}$$

$$N_{exec} = \{n \in N |\ (execute \circ toCode)(n) \neq \emptyset\} \tag{7.12}$$

$$S_{exec} = \{toCode(n), \forall n \in N_{exec}\} \tag{7.13}$$

$$\begin{aligned} objectRepo(c \in C) = \{s \in S_{exec} | \\ (instanceof \circ execute)(s) = c\} \end{aligned} \tag{7.14}$$

Figure 7.2: The core domains and functions of the formal model.

as well as the function used to retrieve snippets for a given class.

As shown in Figure 7.2, $C$ (Equation 7.1) is the domain of classes, $M$ (Equation 7.2) is the domain of methods defined in the classes and $N$ (Equation 7.4) is the domain of AST nodes defined in the methods. Each method is defined in one class (Equation 7.3), and each node is defined in one method (Equation 7.5).

The conversion of the AST nodes into source code is defined as a function *toCode* (Equation 7.7). The codomain of this function is $S$, the domain of all code snippets. Since multiple AST nodes in $N$ can have the same source code representation, *toCode* is a surjective function, and thus for any $N' \subseteq N$ and the corresponding $S'$ it holds that $|S'| \leq |N'|$.

The execution of source code from $S$ is defined as a function

called *execute* (Equation 7.11). Since not all snippets from $S$ will, when executed, yield an object, the codomain of this function is the set $O_0$ (Equation 7.9). This set is defined as the union of object domain (Equation 7.8) and an empty set, used to denote a failed snippet execution. Each object is an instance of a class (Equation 7.10).

With all this in place, we define the set $N_{exec}$ (Equation 7.12) as the set of all AST nodes that, when converted to source code and executed, produce an instance of any class from $C$. We call members of this domain "executable AST nodes". Correspondingly, $S_{exec}$ (Equation 7.13), defines the domain of all "executable" code snippets.

Lastly, the *objectRepo* function (Equation 7.14) returns, for a given class from $C$, a set of snippets from $S_{exec}$ that, when executed, produce an instance of the given class.

### 7.3.2 Implementation

We implemented the approach using EMF. We specify only the back end, leaving open the potential for multiple front ends dealing with the software engineering tasks described in Section 7.2.

Most of the needed implementation was readily available in Pharo *i.e.,* parsing the source code to the AST, converting AST nodes to code snippets, *etc.* The main challenge was implementing the *execute* function

First, we wrap a code snippet inside a closure. We then create a temporary method in a temporary class of which the source code is solely the execution of the mentioned closure. We then compile this method and, if the compilation is successful, we execute it wrapped in the Smalltalk equivalent of a *try-catch* block that catches all possible errors and exceptions.

This setup is enough to catch errors caused by a snippet not being compilable (*e.g.,* containing an undeclared variable) and the snippet failing to execute (*e.g.,* throwing a division by zero exception).

During the execution of code snippets, we encountered some never terminating executions. Further investigation revealed that such issues arise due to concurrency. For example, the snippet might wait on a signal from a different thread to continue the execution. However, we only execute a single snippet at a time, which means there is no chance of receiving such a signal. To restrain such exe-

cutions, we limit the execution time for each snippet to 10 seconds.

The time interval was chosen as an arbitrary cut of point with the reasoning that the execution of any snippet should terminate in less than 10 seconds in order for the snippets to be usable in any way. Although, a vast majority of snippets terminate quite quickly, we chose a very long timeout to include as many snippets whose execution will eventually terminate.

## 7.4 Evaluation

To evaluate our approach we ran it on all classes taken from 141 open source Pharo projects provided by EMF. We do not include all the classes from the base image because a large part of the functionality of the Pharo language is implemented in Pharo itself. Executing code snippets from such classes caused many errors that could not be handled from within the language, but required intervention at the Virtual Machine level. Examples are the contents of packages such of the *Kernel*, *Compiler*, *Debugger*, *NativeBoost* and others.

This evaluation includes a set of classes we call $C'$ containing 13,909 classes. Correspondingly, the set of all methods from $C'$ is noted as $M'$ and contains 256,362 methods. These methods are comprised of 1,525,914 AST nodes defined in a set called $N'$. The number of nodes that are executable is only $|N'_{exec}| = 154,904$ or 10.15% of all the nodes. Converting these nodes into code produces $|S'_{exec}| = 92,460$ unique snippets of code. Table 7.2 presents the cardinalities of these sets.

In the rest of this section we define several sets, shown in Table 7.3, using which we discuss our findings from different perspectives.

We define a set $C_d$ as the domain of the *objectRepo* function as shown in Figure 7.4, Equation 7.15. The cardinality of this set is 10,917 or 78.49% of all classes used in the evaluation. Being able to instantiate almost 80% of all the classes seems to be a promising result considering the minimalistic approach of extracting snippets.

### 7.4.1 Snippet Distribution

We call the set of all classes that can be instantiated through *only one* snippet $C_1$ as shown in Figure 7.4, Equation 7.16. This set helps us to better understand the quality of the snippets, by showing that

| Set | Cardinality |
|---|---|
| $C'$ | 13,909 |
| $M'$ | 256,362 |
| $N'$ | 1,525,914 |
| $N'_{exec}$ | 154,904 (10.15% of $\|N\|$) |
| $S'_{exec}$ | 92,460 |

Table 7.2: The cardinalities of the core sets used in the evaluation.

| Set | Cardinality | $\%\|C'\|$ | $\%\|Cd\|$ | $\%\|C_1\|$ |
|---|---|---|---|---|
| $C_d$ | 10,917 | 78.49% | - | - |
| $C_1$ | 8,779 | 63.12% | 80,42% | - |
| $C_{new}$ | 2,384 | 17.14% | 21.84% | 27.16% |
| $C_l$ | 6,091 | 43.79% | 55.79% | 69.38% |
| $C_g$ | 2,442 | 17.56% | 22.37% | - |

Table 7.3: Cardinalities of the sets defined in Figure 7.4 and their relations.

8,779 classes, or 80,42% of all instantiable classes, have only one associated snippet.

This, plus the fact that $|S'_{exec}| = 92,460$ shows that the distribution of snippet counts is heavily skewed to a minority of classes i.e., about 20%. Further inspection of the snippets shows that the ten classes with the most snippets accumulate over 60% of all snippets. Table 7.4 presents some information about these classes.

Most of the classes from Table 7.4, namely `Array`, `BlockClosure`, `ByteString`, `ByteArray`, `Point` and `Association`, have a very specific construction pattern. Some, like `Array`, `ByteString`, and `ByteArray`, have an idiomatic way of construction. For example, anything in source code between square brackets is considered a `BlockClosure`, anything between single quotation marks is a `ByteString`. Other classes, like `Point` or `Association`, have a very distinctive constructor, *e.g.,* two integers with the @ character between define a point with those integers as coordinates.

With all of this in mind we conclude that the heavy skewing of

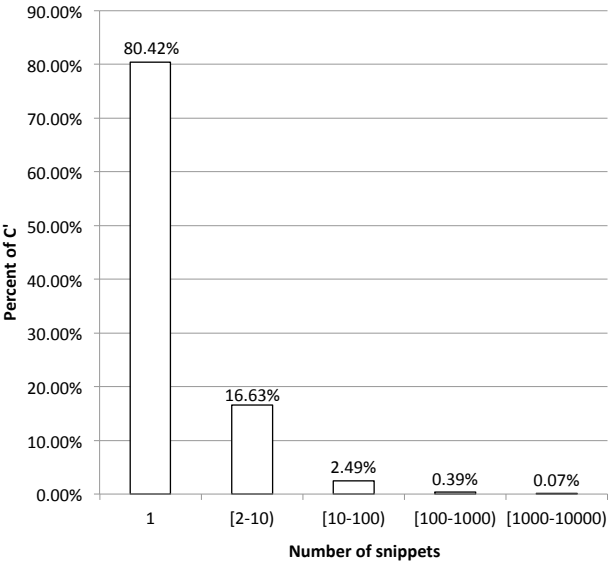


Figure 7.3: Distribution of classes in $C_d$ according to the number of snippets that can instantiate each class.

snippets is not surprising, but it might have a very negative impact on the usability of an Object Repository built with this approach. Although the Object Repository can instantiate almost 80% of all available classes, over 80% of those classes can be instantiated in only one way, and only a handful of classes account for a majority of snippets. In Figure 7.3 we show a distribution of classes in $C_d$ according to how many snippets can instantiate that class. We can see that just under 17% of classes can be instantiated by between two and ten snippets, while less than 3% with ten or more.

### 7.4.2 Trivial and Literal Snippets

To further focus on the quality of the snippets we define the $C_{new}$ set in Figure 7.4, Equation 7.17. This set includes all classes that have only one associated snippet, and that snippet is "trivial" *i.e.,* the default way of creating instances in Pharo. This is achieved by matching snippets with a regular expression that checks if the

| $c \in C'$ | $\lvert objectRepo(c)\rvert$ | $\%\lvert S\rvert$ |
|---|---|---|
| `Array` | 16,907 | 18.29% |
| `ByteString` | 14,968 | 16.19% |
| `BlockClosure` | 8,826 | 9.55% |
| `ByteSymbol` | 4,199 | 4.54% |
| `ByteArray` | 3,807 | 4.12% |
| `Point` | 2,725 | 2.95% |
| `SmallInteger` | 2,502 | 2.71% |
| `Association` | 1,237 | 1.34% |
| `FixedDate` | 855 | 0.92% |
| `Measure` | 759 | 0.82% |
| $\sum$ | 56,785 | 61.42% |

Table 7.4: Ten classes with the most associated code snippets.

snippet is of form "*Class new*". An example of such a snippet would be `Dictionary new` which trivially creates a `Dictionary` object.

The cardinality of this set, as shown in Table 7.3, is 2,384 . This accounts for 27.16% of the classes with a single associated snippet, or 21.84% of all classes from $|C_d|$. Considering that this is the default pattern of instantiating objects in Pharo, the percentage of classes instantiated only in this manner is not as high as might be expected.

We move on to other poor quality snippets by defining the $C_l$ set as shown in Figure 7.4, Equation 7.18. This set is a subset of $C_1$, and contains all classes whose sole associated snippet is just one literal. This set is quite large as can be seen in Table 7.3. It has a cardinality of 6,091 or 55.79% of $C_d$. The size of this set is a result of Smalltalk's high reflective nature. Namely, following the "everything is an object" philosophy, each class in Smalltalk is essentially an instance of a corresponding metaclass, which in turn is an instance of the `Metaclass` class [GR83]. This leads to the phenomenon that executing a class name literal in Smalltalk will result in the object representing that class *i.e.,* an instance of the corresponding metaclass. This phenomenon accounts for all but 8 of the elements of $C_l$ which are global variables mapped to concrete instances of regular (not meta) classes.

$$C_d = \{c \in C' |\ \exists s \in S', objectRepo(s) = c\} \tag{7.15}$$

$$C_1 = \{c \in C_d |\ |objectRepo(c)| = 1\} \tag{7.16}$$

$$C_{new} = \{c \in C_1 |\ \exists s \in objectRepo(c), \\ regexMatch(s, \text{“}\hat{}[a\text{-}zA\text{-}Z0\text{-}9\_]* new\$\text{”})\} \tag{7.17}$$

$$C_l = \{c \in C_1 |\ \exists s \in objectRepo(c), \\ regexMatch(s, \text{“}\hat{}[a\text{-}zA\text{-}Z0\text{-}9\_]*\$\text{”})\} \tag{7.18}$$

$$C_g = C_d \setminus (C_{new} \cup C_l) \tag{7.19}$$

$$C_o = \{c \in C |\ \exists n \in N'_{exec}, (def_m \circ def_n)(n) = c\} \tag{7.20}$$

Figure 7.4: Sets used during the evaluation of the Object Repository.

### 7.4.3 Promising Snippets

An interesting set to focus on is the set of all classes that can be instantiated by the *objectRepo* function in a non-trivial and non-literal way. This is essentially the domain of the *objectRepo* function excluding the sets $C_{new}$ and $C_l$, and is defined as such in Figure 7.4, Equation 7.19. This set, named $C_g$, is actually containing the kind of data we wish to have to realize different use cases introduced in Section 7.2.

As shown in Table 7.3 this set contains 2,442 elements, or 17.56% of all classes included in the evaluation. This is not a large percentage of the classes analysed, but considering the minimalistic approach seems promising as the first step towards realizing the idea of building an Object Repository.

### 7.4.4 Snippet size

The sizes of snippets in the Object Repository varies greatly. The smallest snippets are only one character long, an integer constant producing an instance of `SmallInteger`. The largest snippet is 1,279,918 characters long and is a declaration of a `ByteArray` object. Table 7.5 summarizes the distribution of snippet sizes. We can see by the first quartile (16), median (28) and third quartile (51) that the distribution of the snippet sizes is heavily centred around

| | |
|---|---|
| Minimum | 1 |
| 25% Quartile | 16 |
| Median | 28 |
| 75% Quartile | 51 |
| Maximum | 1,279,918 |

Table 7.5: Five number summary of snippet sizes.

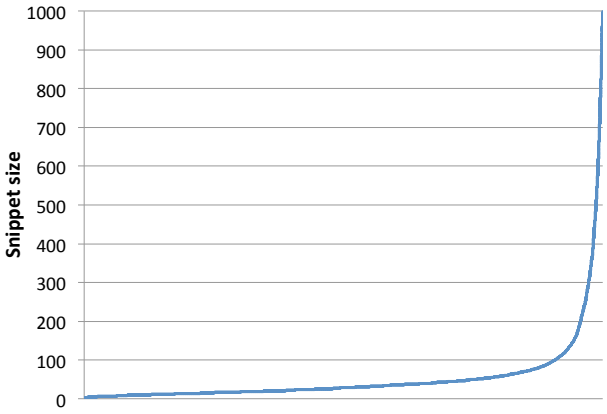


Figure 7.5: A sorted plot of sizes of snippet less than 1000 characters long.

a much more reasonable size.

Since the maximum is so far away from the third quartile, we assumed there are outliers that need to be excluded. But, our attempt to exclude outliers using one and a half times the interquartile range as the limit marked 10.44% of the data as outliers, and we thus include all the data points.

Figure 7.5 demonstrates a sorted plot of all the sizes below 1000, a total of 90,839 snippets or 97.2% of the data set. The remainder of the set was excluded from the plot because the drastic increase in values made the plot very difficult to understand. We can see from the plot that values are mostly under 100, after which a small number of values rises dramatically.

### 7.4.5 Origin of Snippets

To better understand our approach and the resulting snippets we look at where the snippets are coming from. Firstly, we wish to understand how many classes in the $C'$ set actually contributed snippets to the Object Repository. We call these classes "origin classes" and they are members of the $C_o$ set defined in Figure 7.4, Equation 7.20. This set contains 9,138 elements, or 65.70% of $C'$. Manual inspection of a sample of the classes not in this set reveals that they are mostly classes with no or very few declared methods. These are very often meta classes with no functionality outside the trivial instantiation of objects.

We further investigate this set by identifying which classes in this set are meta classes or test classes. We find that meta classes account for 9.78% (894 elements) of $C_o$ which indicates meta classes are less likely to contain snippets, but should not be discarded from the analysis. Test classes account for 16.45% (1,503 elements) of $C_o$. This initially seems to be not much better than the meta classes but adjusting for the number of test classes in $C'$ we can see that the contribution of test classes is much greater. Namely, there is a total of 1,797 test classes in $C'$, which means that over 80% of the available test classes contributed an executable node.

We also find that only 112 classes (1.23% of $C_o$) contributed a snippet that produces an instance of the same class (a snippet that originated in $c \in C'$ and of which the execution results in an instance of $c$). This, coupled with the fact that the remaining 73.77% of $C_o$ are regular classes suggests that the clients of a class are the best place to look for snippets to instantiate that class.

Finally, we aim to answer which types of AST nodes are common sources for snippets. Table 7.6 shows the percentage of occurrences of each type in $N'_{exec}$ with simple, synthetic examples for easier understanding. As one might expect, the most common type of AST node is the *LiteralValueNode*, as it represents a value in the source code and is thus executable by default. The second most common type is the *MessageNode*, which represents sending a message. This is also typical, since in Smalltalk everything happens by sending messages. Third on the list is *VariableNode*, which in our data denotes global variables *i.e.,* meta classes. The remainder of the list can be divided into two categories: wrappers and literals that we discuss in the following.

| Node Type | % | Example snippet |
|---|---|---|
| LiteralValueNode | 33.68% | `'A String'` |
| MessageNode | 22.34% | `Dictionary new.` |
| VariableNode | 17.50% | `Dictionary` |
| SequenceNode | 9.45% | `-` |
| ReturnNode | 7.59% | `^ 'A String'` |
| BlockNode | 6.20% | `[ 1 + 1 ]` |
| LiteralArrayNode | 2.63% | `#(1 2 3)` |
| CascadeNode | 0.34% | `XMLWriter new`<br>`tag: 'one';`<br>`tag: 'two'` |
| ArrayNode | 0.27% | `{1 2 3}` |

Table 7.6: The distribution of types of executable nodes with examples. The *SequenceNode* represents a sequence of other nodes so no example is given.

The wrappers are *SequenceNode*, *ReturnNode* and *CascadeNode*. The *SequenceNode* represents a sequence of nodes. For instance, the node indexed 4 in Figure 7.1 represents a sequence of one node indexed 5 and thus yields the same snippet. The *ReturnNode* just adds the return character[6] in front of the node that it is wrapping. In Smalltalk, the last evaluated expression is returned by default and the return statement may not change the result of executing the snippet. For example, the snippets for *LiteralValueNode* and *ReturnNode* in Table 7.6 have the same execution result. A *CascadeNode* represents a series of message sends to one object. These types of nodes together account for 17.38% of executable nodes.

The literals are *BlockNode*, *LiteralArrayNode* and *ArrayNode*. Closures are very commonly used in Smalltalk and even have their own AST node representation, the *BlockNode*. The other two types of nodes represent a compile time array (*LiteralArrayNode*) and a run time array (*ArrayNode*). These together account for 9.1% of $N'_{exec}$.

[6]The ^ character is the Smalltalk equivalent of the *return* keyword

### 7.4.6 Failed Executions

Studying the reasons why about 90% of AST nodes failed to execute would be necessary towards improving the approach. We primarily hypothesized that a node execution may fail for following reasons:

- Undefined variable in snippet
- Error or exception[7]
- Code snippet returns `nil`

As one might suspect, the execution of the majority of nodes, i.e., over 82%, where prevented because the source code representation of the node failed to compile due to the snippet referring to an undefined variable.

The other two hypothesised faults are not as numerous. Errors and exceptions account for 1,880 AST nodes (0.12% of the total failing AST nodes), and returning nil accounts for 17,425 AST nodes (1.14% of the total failing AST nodes).

Surprisingly an additional 113,954 nodes (8.31% of the total failing ones) fall outside the hypothesised faults. Manual inspection of a sample of the available logs identifies that the main reason for failure was the snippets expecting interactions from the user *e.g.,* opening a dialog for the user to choose a file. Such attempts were immediately shut down due to our code snippets being executed in a headless Pharo environment, meaning that no GUI elements are possible.

### 7.4.7 Missing Classes

To understand why certain classes have no snippets attached to them, we took a sample of 20 such classes and did a manual investigation.

In our sample, 6 classes were test classes. It is not surprising that test classes are never explicitly instantiated, as they are only used by the unit testing framework. Out of all classes with no attached snippets in our data set, around 25% are test classes. Furthermore, 4 classes of our sample were meta classes, and manual inspection shows that these classes, as well as their instances *i.e.,* corresponding

[7]This also includes the nodes terminated by our *timeout* mechanism described in Subsection 7.3.2

non-meta classes, are never used. Looking at all the classes without associated snippets, we find that around 21% are meta classes. The remaining 10 elements of our sample are regular classes, and manual inspection finds that these classes are simply never instantiated. Some are never mentioned in the source code, and some have only class side methods[8] invoked.

None of the sample classes where abstract, but it is understood that all abstract classes would have no snippets associated with them as they, by definition, cannot have instances.

## 7.5 Future Work

We identify several directions for potential future work. The main focus of the future work should be bringing the use cases described in Section 7.2 to fruition, and performing user studies to determine how beneficial the Object Repository would be to developers. This means both improving the approach for building up the object repository, as well as implementing the necessary tools that would serve as the front end facing the developer.

Developing and evaluating these tools is one direction. In Section 7.2 we hypothesized many different tools and approaches and they each raise questions about how useful they would be to the developer and how they would compare to the state of the art.

On the other hand, improving the approach for snippet gathering is also a great challenge. The approach described in this chapter is only a first attempt, with little complexity, and thus, is far from ideal.

In Subsection 7.4.6 we identified that the main reason executing AST nodes fails is that code snippets contain undefined variables. In a statically typed language, this could, to a large extent, be addressed by bootstrapping the Object Repository *i.e.,* using the Object Repository to instantiate all undefined variables at the beginning of the snippet, making all undefined variables not just defined, but instantiated. Lacking static type information would make this not so easily applicable, but still possible through type inference [Mil78, SLN14a, SLN16], or brute force. The downside of this approach is that the instances created would be somewhat synthetic rather than directly pulled from the ecosystem.

[8]The Smalltalk equivalent of static methods in Java

The approach can be improved in many ways by including more static analysis of the source code to build up better snippets before attempting to execute them. Such analysis could be variations on simple compiler analyses [Muc97, ASU86] such as control and data flow analysis, constant propagation, function inlining *etc.*

Also, once we have an approach that is shown to be beneficial to developers, we would like to branch out to different programming languages and examine how different language features such as type systems, reflectivity or mode of execution (compiled vs interpreted) affect the benefits or usability of the Object Repository.

A somewhat related idea to potentially explore is to create a repository of serialised objects at run time. This would require a dynamic analysis approach, through code instrumentation or virtual machine manipulation and would hence be significantly more complex and would introduce run time overhead. On the other hand, this type of approach could give much more representative object instances than the ones mined statically.

## 7.6 Conclusion

In this chapter we propose the idea of an Object Repository, a repository of code snippets that, when executed, produce an instance of a class. We present multiple software engineering tasks that could be improved by the Object Repository.

We further present an initial attempt at implementing the Object Repository, through mining AST nodes and converting them to code snippets. We evaluated the approach on 141 projects written in Pharo Smalltalk, and from the gathered data conclude that our implementation can be used to instantiate almost 80% of all classes encountered in the analysis.

We acknowledge that good data for the proposed Object Repository is for each instantiable class to have multiple ways to be instantiated. Unfortunately, the number of snippets per class is heavily skewed to a small number of classes. In our gathered data, we find that slightly less than 20% of instantiable classes have 2 or more associated snippets. Keeping the simplicity of our approach in mind, this is still a promising result.

We also take a look at the percent of AST nodes that actually produce a type, and discuss the main reasons why other nodes fail

to do so. We find that the main reason for this is undefined variables, accounting for more than 90% of nodes that failed to produce an instance. This is not unexpected, and can be addressed in several ways, including using data from a previous run of the Object Repository itself to instantiate missing variables.

# 8 Conclusion

In this thesis we have argued that it is possible to improve the way ecosystem-aware tools are developed by automating the routine parts, thus freeing the developer to focus on the more important parts of this process: ecosystem data and its presentation. We provide a proof of concept implementation of a framework for developing ecosystem-aware tools, and evaluate its applicability by using it to developing four different tools, each illustrating one aspect of the framework's functionality.

In this chapter we review the main contributions and conclusions drawn throughout the thesis and discuss what we consider to be the most important open questions arising from this work.

## 8.1 A Unified Framework for Ecosystem Aware Tools

The Ecosystem Monitoring Framework fulfills all the requirements to work as a unified framework for developing ecosystem aware tools. It automates all the routine and mundane parts of the development process: defining the ecosystem, loading the source code

for each individual project, executing the user specified analyses as well as storing, providing and periodically refreshing the data that the ecosystem-aware tool needs. Unifying all these steps under one framework ensures that the developer is free from mundane tasks, as well as minimizing the required dependencies.

The full power of EMF can be seen not from EMF itself but from the ecosystem-aware tools that we built using it. The benefits of these tools is clearly shown throughout the thesis, providing significant support for the claim that integrating ecosystem data into developer tools can be beneficial to the developer experience. The type inference approach from Chapter 4 was improved almost 100% by the introduction of ecosystem data which could be reused to help novice developers with unfamiliar APIs as shown in Chapter 5. Developing data supported heuristics for the Type Guesser tool in Pharo was greatly simplified by the reports that our tool from Chapter 6 generated. The Object Repository presented in Chapter 7 opens a whole new dimension of possible tool augmentations by providing objects on demand to any tool developer that might wish to integrate such features into her tool.

These tools are very diverse, each one illustrating one noteworthy feature that EMF provides to tool developers. The ecosystem-aware type inference and the frequently used methods plugin show the clear divide between the back end and front end parts of the tools, by having one back end serve for two different front ends. This is also why we are confident that the Object Repository back end can be used to serve a wide variety of front ends discussed in Section 7.2. The Object Repository itself shows that EMF is not limited to static analysis of source code as text, since the extraction of object-producing code snippets requires the execution of said snippets to verify the type of the result. Since EMF is completely agnostic to the type of analysis the developer wishes to run, any static analysis that a tool developer would need as part of their ecosystem-aware tool is supported, provided that the developer provides the implementation. Finally, as shown by the arguments analysis from Chapter 6, EMF can power tools that have no direct user but rather generate reports on the state of the ecosystem. These reports can further be used to guide development decisions for tools and helping these decisions be supported by the current state of the ecosystem.

Developing these tools individually, by re-implementing the features provided by EMF would have been a much greater develop-

ment effort. Also, as with any code reuse, many other benefits come from relying on one implementation of shared features. The work presented throughout this thesis strongly supports the conclusion that developing ecosystem-aware tools should be done through a unified framework that frees the developer from all the mundane parts of the development, allowing her to focus only on the important parts: the ecosystem data and the way to present it.

## 8.2 Open Questions

This section provides and overview of our picks for the most important open questions arising from this thesis. These questions are outside the scope of this thesis but leave an open space for discussion and future work.

### 8.2.1 Data Freshness

In order to provide fresh data to the front end tools, EMF has to periodically re-run all the analyses on new versions of source code. In our implementation we chose a one week interval, but for some tools and some quickly evolving ecosystems this might not be enough. The main challenge here is that if we wish to have completely fresh data we need to re-run the analyses on every commit to every project in the ecosystem. This is especially problematic if the projects are hosted by a third party (e.g., GitHub, smalltalkhub) which are not willing to give us notifications and full access at every commit, requiring us to poll these repositories for changes. One way this issue could be solved is if the code hosting providers offered a cloud solution for running analyses on the source code. Much like other "infrastructure as a service" solutions this should offer tool developers access to computing machines with preloaded source code of required projects, allowing the developer access to fresh data and providing a source of monetization for the hosting service.

If we manage to obtain every commit in real time there is still the issue that for every commit we need to re-analyze the entire ecosystem. Alternatively, we could express the analyses in such a way as to operate on single commits (i.e., "diffs" in the source code) which would make them much less elegant and understandable. Furthermore we could define the analyses in terms of a model of the ecosystem which is easier to update in real time, relying on source

code only when absolutely necessary. In time, the model would have to be updated as new user needs are identified.

### 8.2.2 Ecosystem Scope

Another important open question is how to define the scope of an ecosystem. In our implementation we relied on a human maintained meta-repository to define our ecosystem. It would be interesting to try to find a way to express certain constraints that would define the ecosystem of interest e.g., all projects using a particular library or framework. This ecosystem definition would allow to automatically extend or shrink the ecosystem scope as the individual projects evolve e.g., adopt or abandon the library we are interested in.

An interesting recent development related to this question is the joint project by GitHub and Google to bring all the GitHub Data to Google's BigQuery data analysis platform[1]. This enables developers to query, using a SQL like query language, all the data on GitHub. This could very well be used as an abstract definition off the scope of an ecosystem, *i.e.,* all projects matching a certain query are part of the ecosystem. The limitation of using BigQuery as the basis for EMF is that the query language provided is quite limited, and not suitable for source code analysis.

Furthermore, we also must ask how much control over the ecosystem scope should the tool developer have. In our implementation of EMF, the ecosystem was pre-defined and all the tools were developed using data from all the ecosystem projects. This might not be a good solution for some tools that require very domain specific data. Allowing the developer to specify which parts of the ecosystem to include or exclude on a per-tool basis might provide better results for some tools.

### 8.2.3 Project History

Throughout this thesis we focused only on providing fresh data to developer tools. This means that we ignore the history of the ecosystem in favor of only acting on the newest versions of the projects. This need not be so, and another version of EMF could offer a way to analyse the source code of all previous versions of the projects as

[1] `https://cloud.google.com/bigquery/public-data/github`

well, opening up the space for a whole new type of ecosystem-aware tools that include historical data.

Another source of historical data is the data gathered by previous runs of EMF. Since EMF allows for unique identification of each execution it is possible, as shown in the class name clash back end example in Listing 3.2, to save the data from each execution in a separate collection. This enables a historical overview of the data gathered by the ecosystem-aware tool and rather then the history of the project. This is important for monitoring the evolution of the tool as well as the impact the tool is having on the ecosystem (*e.g.,* the ecosystem-aware type guessing heuristics could shape the way developers in the ecosystem name their method arguments).

### 8.2.4 Beyond Smalltalk

Much of the simplicity of writing tools with EMF is owed to the reflective nature of Smalltalk. Smalltalk allows us to express both the analyses and the tool front ends in the same general purpose language with little or no need for additional knowledge or tools. In order to port it to more popular languages such as Java we would need a concise and expressive way to analyse the source code of Java projects. This is mainly not trivial for other languages, and we would need to use additional tools (e.g., Moose or Rascal) to analyse the source code. The implications of these additional tools on the ease of development using EMF is very much unknown.

In the future, it might also be worthwhile considering porting EMF to cross language ecosystems e.g., the JVM ecosystem — the ecosystem of all languages that compile to Java byte code. This raises a whole new set of questions and challenges regarding inter-language analysis, mapping concepts from one language to another, etc. Many of these can be addressed by using a language agnostic meta-model (e.g., Famix) but using any model reduces the amount of available information so finding the right level of abstraction is imperative. Projects such as Graal and Truffle [Wue14] aim to bring all languages to one virtual machine, with interlanguage interoperability as a high priority. This might lead to a future where all ecosystems could merge into one mega ecosystem, which would require researchers to rethink the very notion of an ecosystem.

### 8.2.5 Beyond Source Code

Throughout this thesis we develop ecosystem-aware tools relying only on source code of the projects in the ecosystem. Software development today produces a wide range of different artifacts that supplement the source code and are used as additional data sources for ecosystem-aware tools. Including this additional data into EMF would open new possibilities for developing different kinds of ecosystem-aware tools, but also raise a series of challenges on how to obtain, analyze or keep that data fresh.

### 8.2.6 Beyond Our Tools

Finally, we still need a comprehensive study on user needs when it comes to ecosystem-aware tools. The tools we developed were based on our own intuition and the need to demonstrate the features of EMF. Even though our analyses show that ecosystem data improves these tools, future work would be best served by finding exact user needs, and ensuring that EMF can support tools that tackle those problems. The work of Haenni *et al.* [HLSN13, HLSN14] would be a good foundation for defining exact user needs.

## 8.3 Summary

We propose a unified framework for developing ecosystem-aware tools. By automating the routine parts of the process (defining the ecosystem scope, loading individual projects, executing analyses, gathering and providing the results, *etc.*) we enable the developer to focus only on answering the two important questions for an ecosystem-aware tool – what data is needed from the ecosystem and how should it be presented? A proof of concept framework is implemented and four different ecosystem-aware tools where developed to illustrate that this type of framework is viable. Each of the tools was evaluated resulting in several noteworthy observations.

The first tool we implemented was a type inference engine relying on measuring the frequency of association between a message and a type throughout the ecosystem source code. We find that this data is helpful in identifying correct types and that it leads to a substantial increase in the number of correctly inferred types.

The second tool aims to improve the time a developer spends browsing documentation in order to find the name of a method for a particular functionality. We conducted a case study which shows that most API classes have only a small number of methods which are frequently invoked. Guided by this observation we augmented a documentation browser to show the most frequently invoked methods of the current class, thus aiding the developer to, in most cases, quickly find the method of interest.

We further focused on type guessing, the process of concluding the type of a method argument based on its name. We developed a tool that generates a report on the state of the argument names in the ecosystem. We used the report to formulate several new heuristics for guessing types and also concluded that at least one in two method arguments in Smalltalk projects contains a useful type hint and that 1.44% of method arguments hint at multiple types. We compared our guesses with run-time types and found that in 76% of cases the type hint matches the run-time type.

And finally we mined the ecosystem for code snippets that produce objects in order to support several hypothesised tools. We found that even a naive approach can extract snippets for almost 80% of the analyzed classes, but the snippets are of very poor quality. We also conclude that the best place to mine snippets that instantiate a class is in the clients and the tests of the class.

# Bibliography

[ABC+13] Vanessa Peña Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, September 2013.

[ACF+13] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, August 2013.

[ADG04] Omar Alonso, Premkumar Devanbu, and Michael Gertz. Database techniques for the analysis and exploration of software repositories. *IET Conference Proceedings*, pages 37–41(4), jan 2004.

[Age95] Ole Agesen. The Cartesian product algorithm. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 2–26, Aarhus, Denmark, August 1995. Springer-Verlag.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, Mass., 1986.

[AXPX07] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM.

[BB01] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001.

[BBS10] Jan Bosch and Petra Bosch-Sijtsema. From integration to composition: On the impact of software product lines, global development and ecosystems. *J. Syst. Softw.*, 83(1):67–76, January 2010.

[BDN+09] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[BGR+09] Philipp Bunge, Tudor Gîrba, Lukas Renggli, Jorge Ressia, and David Röthlisberger. Scripting browsers with Glamour. European Smalltalk User Group 2009 Technology Innovation Awards, August 2009. Glamour was awarded the 3rd prize.

[BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[BOL09] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, ICSE Workshop on*, 0:1–4, 2009.

[BOL14] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.*, 79:241–259, January 2014.

[BW12] Raymond P. L. Buse and Westley Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, Piscataway, NJ, USA, 2012. IEEE Press.

[BWKG05] Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 177–186, New York, NY, USA, 2005. ACM.

[CCSL14] Andrea Caracciolo, Andrei Chiş, Boris Spasojević, and Mircea Lungu. Pangea: A workbench for statically analyzing multi-language software corpora. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 71–76. IEEE, September 2014.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM.

[CGNS15] Andrei Chiş, Tudor Gîrba, Oscar Nierstrasz, and Aliaksei Syrel. GTInspector: A moldable domain-aware object inspector. In *Proceedings of the Companion Publication of the 2015 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH Companion 2015, pages 15–16, New York, NY, USA, 2015. ACM.

[CKMN03] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng. Mirror adaptive random testing. In *Proceedings of the Third International Conference on Quality Software*, QSIC '03, pages 4–, Washington, DC, USA, 2003. IEEE Computer Society.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DGGH11] Julius Davies, Daniel M. Germán, Michael W. Godfrey, and Abram Hindle. Software bertillonage: Finding the provenance of an entity. In *MSR'11: Proceedings of the 8th International Working Conference on Mining Software Repositories*, pages 183–192, 2011.

[DLR07] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

[DLR11] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, pages 1–47, 2011.

[DMCG16] A. Decan, T. Mens, M. Claes, and P. Grosjean. When github meets cran: An analysis of inter-repository package dependency problems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 493–504, March 2016.

[DN81] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.

[DNRN13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.

[DP03] Dirk Draheim and Lukasz Pekacki. Process-centric analytical processing of version control data. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 131–136, Los Alamitos CA, 2003. IEEE Computer Society Press.

[DPGA10] M. Di Penta, D.M. German, and G. Antoniol. Identifying licensing of jar archives using a code-search ap-

proach. In *Mining Software Repositories (MSR), 2010 7th IEEE WorkingConference on*, pages 151–160, May 2010.

[GGMT14] Mohammad Ghafari, Carlo Ghezzi, Andrea Mocci, and Giordano Tamburrelli. Mining unit tests for code recommendation. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 142–145, New York, NY, USA, 2014. ACM.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Addison Wesley, Reading, Mass., May 1983.

[Han10] Stefan Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. *SIGPLAN Not.*, 45(10):22–35, October 2010.

[HKV12] Mark Hills, Paul Klint, and Jurgen J. Vinju. Scripting a refactoring with rascal and eclipse. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 40–49, New York, NY, USA, 2012. ACM.

[HLSN13] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proceedings of the 1st Workshop on Ecosystem Architectures*, pages 1–5, 2013.

[HLSN14] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. A quantitative analysis of developer information needs in software ecosystems. In *Proceedings of the 2nd Workshop on Ecosystem Architectures (WEA'14)*, pages 1–6, 2014.

[HM05] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[Hol06] Reid Holmes. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, 2006.

[JFB09] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. A sense of community: A research agenda for software ecosystems. In *Presented at the 31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009, IEEE*, pages 187–190, 2009.

[KBR14] Juraj Kubelka, Alexandre Bergel, and Romain Robbes. Asking and answering questions during a programming change task in the Pharo language. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '14, pages 1–11, New York, NY, USA, 2014. ACM.

[KDV07] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[KLHK09] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Adding examples into java documents. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 540–544, Washington, DC, USA, 2009. IEEE Computer Society.

[KLR+13] Jan Kurš, Guillaume Larcheveque, Lukas Renggli, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. PetitParser: Building modular parsers. In *Deep Into Pharo*, page 36. Square Bracket Associates, September 2013.

[KRG+14] R. G. Kula, C. D. Roover, D. German, T. Ishio, and K. Inoue. Visualizing the evolution of systems and their library dependencies. In *2014 Second IEEE Working Conference on Software Visualization*, pages 127–136, sep 2014.

[LLGR10] Mircea Lungu, Michele Lanza, Tudor Gîrba, and Romain Robbes. The Small Project Observatory: Visualizing software ecosystems. *Science of Computer Programming, Elsevier*, 75(4):264–275, April 2010.

[LM10a] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 185–194, New York, NY, USA, 2010. ACM.

[LM10b] Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 8:1–8:6, New York, NY, USA, 2010. ACM.

[LPS11] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1317–1324, New York, NY, USA, 2011. ACM.

[LRL10] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *ASE'10: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM Press, 2010.

[Lun09] Mircea Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, November 2009.

[Man16] Konstantinos Manikas. Revisiting software ecosystems research: A longitudinal literature study. *Journal of Systems and Software*, 117:84 – 103, 2016.

[MBFV13] João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *WCRE*, pages 401–408, 2013.

[MBGN16] Nevena Milojković, Clément Béra, Mohammad Ghafari, and Oscar Nierstrasz. Inferring types by mining class usage frequency from inline caches. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2016)*, pages 6:1–6:11, 2016.

[MDZ10] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. Mining api popularity. In *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*, TAIC PART'10, pages 173–180, Berlin, Heidelberg, 2010. Springer-Verlag.

[MH13] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems — a systematic literature review. *J. Syst. Softw.*, 86(5):1294–1306, May 2013.

[MHR+12] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. *SIGPLAN Not.*, 47(10):683–702, October 2012.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MMS01] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27(12):1085–1110, December 2001.

[MN16] Nevena Milojković and Oscar Nierstrasz. Exploring cheap type inference heuristics in dynamically typed languages. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 43–56, New York, NY, USA, 2016. ACM.

[MS03] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry.* MIT Press, Cambridge, MA, USA, 2003.

[MS05] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry.* The MIT Press, 2005.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[MW03] J. L. Myers and A. D. Well. *Research Design and Statistical Analysis*. Lawrence Erlbaum Associates, New Jersey, 2003.

[MXBK05] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. *SIGPLAN Not.*, 40(6):48–61, June 2005.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, September 2005. ACM Press. Invited paper.

[NNP+09] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[OAC+06] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, École Polytechnique Fédérale de Lausanne, 2006.

[OBL10] J. Ossher, S. Bajracharya, and C. Lopes. Automated dependency resolution for open source software. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 130 –140, may 2010.

[PB13] Zvonimir Pavlinović and Domagoj Babić. Interactive code snippet synthesis through repository mining. Technical Report UCB/EECS-2013-23, EECS Department, University of California, Berkeley, mar 2013.

[PJAG12] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 925–935, Piscataway, NJ, USA, 2012. IEEE Press.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[PMW09] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[RBK+13] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013.

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74:470–495, May 2009.

[Riv96] Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996.

[RLP13] C. De Roover, R. Laemmel, and E. Pek. Multi-dimensional exploration of api usage. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 152–161, may 2013.

[Sal04] Michael Salib. Faster than C: Static type inference with Starkiller. In *in PyCon Proceedings, Washington DC*, pages 2–26. SpringerVerlag, 2004.

[Sca06] Christopher Scaffidi. Why are APIs difficult to learn and use? *Crossroads*, 12(4):4–4, August 2006.

[SGN16] Boris Spasojević, Mohammad Ghafari, and Oscar Nierstrasz. The object repository, pulling objects out of the ecosystem. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*, IWST'16, pages 7:1–7:10, New York, NY, USA, 2016. ACM.

[SLN14a] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. Mining the ecosystem to improve type inference for dynamically typed languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '14, pages 133–142, New York, NY, USA, 2014. ACM.

[SLN14b] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. Overthrowing the tyranny of alphabetical ordering in documentation systems. In *2014 IEEE International Conference on Software Maintenance and Evolution (ERA Track)*, pages 511–515, September 2014.

[SLN14c] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. Towards faster method search through static ecosystem analysis. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ECSAW '14, pages 11:1–11:6, New York, NY, USA, August 2014. ACM.

[SLN16] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. A case study on type hints in method argument names in Pharo Smalltalk projects. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 283–292, March 2016.

[TAD+10] E. Tempero, C. Anslow, J. Dietrich, T. Han, Jing Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336 –345, December 2010.

[Tes81] Larry Tesler. The Smalltalk environment. *Byte*, 6(8):90–147, August 1981.

[TFH09] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 3rd edition, 2009.

[TX07] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[TX08] S. Thummalapenta and Tao Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 327–336, Washington, DC, USA, 2008. IEEE Computer Society.

[VLBN09] Rajesh Vasa, Markus Lumpe, Philip Branch, and Oscar Nierstrasz. Comparative analysis of evolving software systems using the Gini coefficient. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, pages 179–188, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[WHH11] A. Wiese, V. Ho, and E. Hill. A comparison of stemmers on source code identifiers for software search. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 496–499, September 2011.

[Whi12] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, third edition edition, May 2012.

[Wue14] Thomas Wuerthinger. Graal and truffle: Modularity and separation of concerns as cornerstones for building a multipurpose runtime. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 3–4, New York, NY, USA, 2014. ACM.

[ZXZ+09] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns.

In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 318–343. Springer Berlin Heidelberg, 2009.

[ZYZ+12] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 826–836, Piscataway, NJ, USA, 2012. IEEE Press.